MA0122–00

68xxx v9.0

InterTools C Compiler/Assembler User's Guide

TASKING

Quality Development Tools Worldwide

A publication of
TASKING
Documentation Department
Copyright © 1997 TASKING, Inc.

All rights reserved. Reproduction in whole or part is prohibited without the written consent of the copyright owner.

The following trademarks are acknowledged:

DOS16M and DOS/4GW are trademarks of Rational Systems, Inc.
FLEXlm is a registered trademark of Globetrotter Software, Inc.
HP and HP–UX are trademarks of Hewlett–Packard Co.
IBM is a trademark of International Business Machines Corp.
Motorola is a trademark of Motorola, Inc.
MS–DOS and Windows are registered trademarks of Microsoft Corporation.
Solaris is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.
VAX and VMS are trademarks of Digital Equipment Corporation.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

E–mail: support@tasking.com
WWW: http://www.tasking.com

The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, TASKING assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

TASKING reserves the right to change specifications embodied in this document without prior notice.

Table of Contents

CHAPTER 1 1–1

Introduction 1–1

1.1 InterTools Documentation 1–1

1.1.1 The InterTools Manuals 1–3

1.1.2 Documentation Conventions 1–4

CHAPTER 2 2–1

68000 Family C Compiler 2–1

2.1 Introduction 2–1

2.2 C Compiler Options: Summary 2–2

2.3 Usage 2–10

2.4 C Compiler Options: Detailed Descriptions 2–12

2.4.1 Listing Options 2–12

2.4.2 Include Options 2–16

2.4.3 Datatype Options 2–19

2.4.4 Separate Data Options 2–26

2.4.5 Optimizer Options 2–29

2.4.6 Floating Point Options 2–39

2.4.7 Code Generation Options 2–44

2.4.8 Position–independent Code Options 2–57

2.4.9 Miscellaneous Options 2–62

2.5 Using the Optimizer 2–71

2.6 Optimizations Performed 2–73

2.6.1 Automatic Register Variable Assignment 2–73

2.6.2 Common Subexpression Elimination 2–74

2.6.3 Target Path Computation 2–75

2.6.4 Strength Reduction 2–76

2.6.5 Code Hoisting 2–76

2.6.6 Loop Rotation 2–77

2.6.7 Branch Tables 2–78

2.6.8 Entry/Exit Optimization 2–78
2.6.9 Multiplication Optimization 2–79
2.6.10 Subscript Optimization 2–79
2.6.11 Special Instruction Selection 2–80
2.6.12 Special Addressing Modes 2–80
2.7 Messages 2–80

CHAPTER 3 3–1
68000 Family Assembler 3–1
3.1 Introduction 3–1
3.2 Assembler Options: Summary 3–2
3.3 Usage 3–6
3.4 Assembler Options: Detailed Descriptions 3–7
3.4.1 Listing Options 3–7
3.4.2 INCLUDE Options 3–14
3.4.3 Code Generation Options 3–15
3.4.4 Miscellaneous Options 3–17
3.5 Messages 3–19

CHAPTER 4 4–1
Linking Locator 4–1
4.1 Introduction 4–1
4.2 Linking Locator Options: Summary 4–2
4.3 Usage 4–4
4.3.1 Linking 4–4
4.3.2 ROM Processing 4–5
4.3.3 Locating 4–7
4.4 Linking Locator Options: Detailed Descriptions 4–8
4.4.1 Linker Options 4–9
4.4.2 Locator Options 4–10
4.4.3 ROM Processing Options 4–11
4.4.4 Symbol Options 4–14
4.4.5 Miscellaneous Options 4–16
4.5 Linking Concepts 4–19
4.5.1 Segments 4–19
4.5.2 Groups 4–22
4.5.3 Classes 4–24

4.5.4 Relocation 4–26
4.6 Compiler Library Organization 4–27
4.7 Library Searches 4–33
4.8 Locator Commands 4–34
4.8.1 General Command Syntax 4–35
4.8.2 Comments 4–35
4.8.3 Numbers 4–36
4.8.4 Keywords 4–36
4.8.5 Address Ranges 4–36
4.8.6 Names 4–37
4.8.7 Name List 4–37
4.9 Command Descriptions 4–37
4.10 Messages 4–49

CHAPTER 5 5–1
Formatter 5–1
5.1 Introduction 5–1
5.2 Formatter Options: Summary 5–2
5.3 Usage 5–6
5.3.1 form 5–6
5.3.2 form695 5–7
5.4 Formatter Options: Detailed Descriptions 5–7
5.4.1 Format Options 5–8
5.4.2 PROM Options 5–16
5.4.3 COFF Format Options 5–19
5.4.4 CrossView Pro Options 5–20
5.4.5 Miscellaneous Options 5–23
5.5 IEEE–695 Formatter Limitations 5–26
5.6 Messages 5–27

CHAPTER 6 6–1
Other Utilities 6–1
6.1 Librarian 6–3
6.1.1 Librarian Options: Summary 6–3
6.1.2 Usage 6–5
6.1.3 Librarian Options: Detailed Descriptions 6–7
6.2 Global Symbol Mapper 6–13

6.2.1 Global Symbol Mapper Options: Summary 6–14
6.2.2 Usage 6–15
6.2.3 Global Symbol Mapper Options: Detailed Descriptions 6–17
6.2.4 Messages 6–20
6.3 Symbol List Utility 6–21
6.3.1 Symbol List Utility Options: Summary 6–22
6.3.2 Usage 6–22
6.3.3 Symbol List Utility Options: Detailed Descriptions 6–23
6.3.4 The Symbol Table Listing 6–24
6.3.5 Messages 6–27
6.4 Object Size List Utility 6–29
6.4.1 Object Size List Utility Options: Summary 6–29
6.4.2 Usage 6–30
6.4.3 Object Size List Utility Options: Detailed Descriptions 6–30
6.4.4 Messages 6–32

CHAPTER 7 7–1

Application Notes 7–1
7.1 About the Application Notes 7–1
7.2 Downloading 7–5
7.2.1 Introduction 7–5
7.2.2 PROM Programming 7–6
7.3 Linking C and Assembly 7–9
7.3.1 Introduction 7–9
7.3.2 Conventions 7–10
7.3.3 Sharing Global Data 7–14
7.4 Pragma Separate (Option Separate) 7–17
7.4.1 Introduction 7–17
7.4.2 Preprocessor Option Directives 7–18
7.4.3 Command Line Switches 7–20
7.5 Building Libraries That Do Not Use A5 7–21
7.6 Position–independent Code 7–37
7.6.1 Introduction 7–37
7.6.2 How Position Independence is Achieved 7–38
7.6.3 Position Independence and Data References 7–41
7.6.4 Position Independence and Data Initialization 7–47
7.6.5 Building a Position–independent System 7–48

7.6.6 Some Additional Hints 7–52
7.7 Getting the Best Code for Your Application 7–53
7.7.1 Code Size versus Execution Speed 7–53
7.7.2 If Statements 7–54
7.7.3 Using Integer Data 7–54
7.7.4 Size of int Datatype 7–56
7.7.5 Compilation Models for Data 7–59
7.8 Support for the On–board Peripherals 7–63
7.9 InterTools FLEXlm Guide (Unix only) 7–69
7.9.1 License Administration 7–69
7.9.2 FLEXlm User Commands 7–76
7.9.3 FLEXlm Runtime Error Messages and Codes 7–87
7.9.4 The Daemon Log File 7–97
APPENDIX A A–1
C Language Specifications A–1
1 Introduction A–1
2 Preprocessor Extensions A–3
3 In–line Assembly Language A–3
3.1 The _CASM method A–7
3.2 The _ASM method A–9
3.3 Syntax Summary A–13
4 ANSI C Function Prototypes A–16
4.1 Creating Function Prototypes A–16
4.2 Calls to Functions with Prototypes A–19
5 Other ANSI C Features A–21
6 Adjacent String Literal Concatenation A–21
8 Void Pointers – void * A–23
9 Const Type Qualifier A–23
10 Stringization A–25
11 ANSI C Preprocessor Additions A–26
11.1 New Predefined Macros A–26
11.2 New Directives A–27
11.3 #error A–27
11.4 #pragma A–27
11.5 #elif A–27
12 Volatile Type Qualifier A–28
13 New Operators A–29

13.1 defined A–29
13.2 token pasting A–30
14 Support for Interrupt Handlers in C A–31
14.1 The _GPL Pseudo–Function A–33
14.2 The _SPL Pseudo–Function A–33
14.3 The _TRAP Function A–34
14.4 The _IH Keyword A–34
14.5 The _SWI Keyword A–36
15 Implementation–Defined Behavior A–37
APPENDIX B B–1
Compiler Naming Conventions B–1
1 Introduction B–1
2 Code Symbols B–2
3 Data Symbols B–3
3.1 Global Data B–3
3.2 Local Static Data B–4
3.3 Stack Data B–4
3.4 String Constants B–4
3.5 Other Symbols B–4
4 Segment Names B–5
4.1 Code Segment Names B–5
4.2 Data Segment Names B–5
4.3 Separate Data B–6
5 Symbol Naming Summary B–8
5.1 Notes B–9

APPENDIX C **C–1**
Compiler Runtime Conventions C–1
1 Introduction C–1
2 Storage Allocation C–1
2.1 Notes C–2
3 Segmentation Model C–2
4 Register Usage C–4
5 Subroutine Linkage C–5
5.1 Preserved Registers C–5
5.2 Register Return Values C–5
5.3 Parameter Passing C–5
5.4 Calling Sequence C–6
5.5 Procedure Prologue C–6
5.6 Initial Startup C–9

APPENDIX D **D–1**
Object Module Formats D–1
1 Introduction D–1
2 Intel ASCII Hex Format D–2
3 Motorola S Records D–3
4 Extended Motorola S Records D–4
5 Packed Motorola S Records D–5
6 S37 Motorola S Records D–6
7 Tektronix Format (Tekhex) D–7
8 Extended Tekhex Format D–7
8.1 Section Definition Field D–10
8.2 Symbol Definition Field D–11
9 Binary Tektronix Format D–11
10 HP64000 Format D–12
10.1 Using the HP64000 Format D–12
10.2 Files Needed D–14
10.3 Generating Files for Use with the 64700 ..... D–15
10.4 Formatter Examples for PC and Unix Hosts D–16
10.5 Using get64 on Unix Hosts D–17
10.6 InterTools and get64 on Unix Hosts D–17
11 Common Object File Format (COFF) D–19
11.1 File Header D–19
11.2 Option Header D–21
11.3 Relocation Information D–21
11.4 Section Headers D–21
11.5 Line Number Information D–21
11.6 Symbol Table Entries D–22
11.7 COFF1 Format D–22
APPENDIX E **E–1**
Compiler/Assembler Driver E–1
1 For PC and Unix Hosts: E–1
APPENDIX F **F–1**
Supplemental Reading F–1
1 Bibliography F–1

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is for users of the InterTools 68xxx C compiler/assembler.

MANUAL STRUCTURE

1. Introduction

Introduces the documentation conventions and organization.

2. 68000 Family Compiler

Describes the operation and use of the InterTools 68xxx C Compiler, including options, optimizer options, and error messages.

3. 68000 Family Assembler

Describes the operation and use of the InterTools Assembler.

4. Linking Locator

Describes the operation and use of the Linking Locator utility, including options, linking concepts, compiler run–time libraries, library searches, locator commands, and error messages.

5. Formatter

Describes the operation and use of two formatter utilities, `form` and `form695`, including options and error messages.

6. Other Utilities

Describes the following utilities: Librarian, Global Symbol Mapper, Symbol List Utility, and Object Size Utility.

7. Application Notes

Contains information on the following topics:

- Downloading
- Linking C and Assembly
- Pragma Separate (Option Separate)
- Building Libraries that do not use A5
- Position–independent Code
- Getting the best code for your application
- Support for the on–board peripherals of the 68332, 68340, and 68360
- InterTools FLEXlm Guide (Unix only)

APPENDICES

A. C Language Specifications

Contains information on the following:

- preprocessor extensions
- in–line assembly
- ANSI C function prototypes
- the const type qualifyer
- implementation–defined behavior

B. Compiler Naming Conventions

Contains information on the following:

- Code Symbols
- Data Symbols
- Segment Names

Contains a Symbol Naming Summary.

Describes Storage Allocation, the Segmentation Model, Register Usage, Subroutine Linkage, Stack Layout, and Initial Startup.

C. Compiler Runtime Conventions

Describes Storage Allocation, the Segmentation Model, Register Usage, Subroutine Linkage, Stack Layout, and Initial Startup.

D. Object Module Formats

Describes all of the various object module formats.

E. Compiler/Assembler Driver

Discusses the driver that invokes the Compiler and Assembler executables.

F. Supplemental Reading

A listing of information sources of interest to InterTools 68xxx C compiler/assembler users.

CHAPTER 1

Introduction

This *InterTools User's Manual* contains invocation, options, and usage summaries, along with examples for each of the tools and definitions of special terminology and functions. The structure of all InterTools documentation and conventions is described in this chapter. Please refer to the *Introduction* in the *InterTools Getting Started Manual* for information concerning the InterTools system and for additional help.

1.1 InterTools Documentation

The InterTools documentation is composed of the *InterTools Getting Started Manual*, the *InterTools User's Manual*, and the *InterTools Reference Manual*. Their contents are represented in the following three charts:

InterTools Getting Started Manual
1. Introduction 2. Installation Guide 3. InterTools Tutorial

InterTools User's Manual

1. Introduction
2. 68000 Family C Compiler
3. 68000 Family Assembler
4. Linking Locator
5. Formatter
6. Other Utilities:
   - Introduction
   - Librarian
   - Global Symbol Mapper
   - Symbol List Utility
   - Object Size List Utility
7. Application Notes:
   - Introduction
   - Downloading
   - Linking C and Assembly
   - Pragma Separate (Option Separate)
   - Building Libraries That Do Not Use A5
   - Position–independent Code
   - Getting the Best Code For Your Application
   - Support for the On–board Peripherals of the 68332, 68340, and 68360
   - InterTools FLEXlm Guide (Unix only)

A. Appendix: C Language Specifications
B. Appendix: Compiler Naming Conventions
C. Appendix: Compiler Run Time Conventions
D. Appendix: Object Module Formats
E. Appendix: Compiler/Assembler Driver
F. Appendix: Supplemental Reading

InterTools Reference Manual
1. Introduction
2. Run–time Library
3. The Assembly Language Reference
4. Source Program Coding
5. Assembler Directives
6. Macro Operations and Conditional Assembly
7. Structured Control Statements
8. Position–independent Code
A. Appendix: Instruction Set Summary
B. Appendix: Character Set

1.1.1 The InterTools Manuals

The *InterTools Getting Started Manual* contains an introduction to the InterTools system and customer support, an installation guide, and a tutorial which contains sample code and exercises which lead you step–by–step through the powerful features of each software tool.

The *InterTools User's Manual* includes invocation, options, and usage summaries, along with examples for each of the tools and definitions of special terminology and functions. This manual also contains additional information in the appendices on run time and naming conventions, C language extensions, and object module formats,

The *InterTools Reference Manual* provides information on the run–time libraries and the information necessary to write programs in assembly language. It contains sections on source program coding, assembler directives, macro operations, structured control statements, and position–independent code, as well as a summary of the target instruction set and character set.

1.1.2 Documentation Conventions

In this documentation set, we use a number of typefaces to demonstrate the syntax of various commands and to show sample text you might type at a terminal or observe in a file. The following is a list of these conventions.

Typewriter Font

Used for: user input/screen output

`Typewriter` font is used in examples to show what you might type at a terminal: command names, directives, switches, literal filenames, or any other text which must be typed exactly as shown. It is also used in other examples to show what you might see on a screen or in a printed listing. For example:

```
form [prog.ab] [options]
```

`form` is shown in typewriter font because it must be typed exactly as shown when you run the formatter.

```
#include "stdio.h"
main ()
{
   printf ("hello, world.\n");
}
```

The above example shows the contents of a file that you may have output to the screen.

Italics

Used for: value substitution

Italic type indicates categories of items for which you must substitute appropriate values, such as arguments or hypothetical filenames. For example:

```
-c cfn
```

`-c` is in typewriter font because it must be typed exactly as shown. *cfn* is in italics, indicating that you must substitute an appropriate value, in this case the name of a locator command file, in place of *cfn* when using this option.

We also use *italic* type throughout this documentation set to indicate directory names in explanations.

Square Brackets

Used for: optional items

Optional items in syntax, command, or option expressions appear in square brackets ('[' and ']'). As an example, the syntax of the `-w` assembler option is shown below:

```
-w [n]
```

The *n* is optional and thus is enclosed in square brackets.

Vertical Bar

Used for: indicating that only one of the items in the group may be chosen

A vertical bar separating two or more items indicates that a choice must be made among the items. For example:

```
llink prog.[ol|rmp|ln]
```

The user should select one of the filename extensions shown.

Ellipsis

Used for: indicating that one or more items are allowed

An ellipsis (...) is used after an item to indicate that more than one of that item is allowed. The syntax of the `-I` compiler option is shown below:

```
-I dir1 [dir2...]
```

The `-I` must be typed exactly as shown. A directory name must be substituted for *dir1* and after that there may optionally be any number of additional directory names.

Shaded Boxes

Used for: command line examples

Command line examples appear in shaded boxes, as below:

```
llink prog.ol
```

Bounded Boxes

Used for: listing output

Bounded boxes display listings, as below:

```
January 20, 1997 CROSS-REFERENCE: hello.c PAGE 1

i
   Def : hello.h      2
   Ref : hello.c      5

main
   Def : hello.c      2
```

Note: Due to the page width limitations of this manual, we may represent a single invocation line as two or more lines. You should, however, type the invocation on one line.

CHAPTER 2

68000 Family C Compiler

This chapter describes the operation and use of the 68000 Family C Compiler. It begins with a summary listing of the available options and continues with more detailed explanations of their usage, the optimizer functions, and error messages.

2.1 Introduction

Function: Compile C program(s) for the following targets:

Target	Invocation
MC68000	c68000 *prog*.c [*prog2*.c ...] [*options*]
MC68EC000	c68000 *prog*.c [*prog2*.c ...] [*options*]
MC68008	c68000 *prog*.c [*prog2*.c ...] [*options*]
MC68010	c68010 *prog*.c [*prog2*.c ...] [*options*]
MC68020	c68020 *prog*.c [*prog2*.c ...] [*options*]
MC68EC020	c68020 *prog*.c [*prog2*.c ...] [*options*]
MC68030	c68030 *prog*.c [*prog2*.c ...] [*options*]
MC68EC030	cec030 *prog*.c [*prog2*.c ...] [*options*]
MC68040	c68040 *prog*.c [*prog2*.c ...] [*options*]
MC68EC040	cec040 *prog*.c [*prog2*.c ...] [*options*]
MC68302	c68302 *prog*.c [*prog2*.c ...] [*options*]
MC68330	c68332 *prog*.c [*prog2*.c ...] [*options*]

Target	Invocation
MC68331	c68332 *prog*.c [*prog2*.c ...] [*options*]
MC68332	c68332 *prog*.c [*prog2*.c ...] [*options*]
MC68340	c68340 *prog*.c [*prog2*.c ...] [*options*]
MC68360	c68360 *prog*.c [*prog2*.c ...] [*options*]
MC68060	c68060 *prog*.c [*prog2*.c ...] [*options*]

Input: *prog*`.c` [*prog2*`.c` ...]

Output: *prog*`.ol` [`.lis, .pp, .psa .xrf, .s`] [*prog2*`.ol`...]

2.2 C Compiler Options: Summary

The C compiler recognizes the following options:

Option	Function	See:
`-0`	(Zero) Display the version number of the executables.	2–62
`-68`	Software floating point compatibility mode. This option has no effect unless the `-h` option is also supplied. **WARNING**: This switch is not compatible with routines in the standard run–time library that return doubles.	2–39
`-a`	Generate source listing and show included source.	2–12
`-aa`	Align each procedure on a 16–byte boundary. May be useful on 68040 to optimize instruction cache.	2–44

Option	Function	See:
`-ab`	Force word alignment for structures containing bit fields.	2–20
`-ac`	Align only the first procedure on a 16–byte boundary.	2–45
`-ai`	Expand procedures inline.	2–29
`-ao` *options*	Pass specified options to the assembler step. Options must be in quotes if there's more than one given. Use only with `-ia`.	2–62
`-ar`	Use alternate register usage conventions. **WARNING:** The switch is not compatible with the standard run–time library.	2–47
`-b5`	Use 32–bit A5–relative offsets.	2–47
`-bb`	Maintain backwards compatible bitfield storage.	2–21
`-C`	Old run–time model compatibility mode. Older compiler preserved fewer registers across procedure calls.	2–49
`-ca`	Continue compilation to completion, even if `-E` or `-M` is present.	2–62
`-cc` *classname*	Set class of generated code segment to *classname*.	2–50
`-cs`	Put data declared with the `const` type qualifier into a **separate** segment *cdata* and class *constant*.	2–26
`-D` *tbs* [*tbs* ...]	Define datatype *t* as *b* bytes with sign *s*.	2–21

Option	Function	See:
`-d`	Generate symbolic debugging information.	2–62
`-dd`	Allow ANSI–style duplicate declarations. **Note:** This switch changes the order of allocation for uninitialized global variables. Programs which depend on global variables being allocated one after another must not use this switch.	2–62
`-do`	Disable all optimizations which interfere with debugging. This is equivalent to `-nd`, `-nh`, `-nl`, `-np` and `-nr`.	2–32
`-E` [*pfn*]	Save preprocessor output in file *pfn*. If *pfn* is omitted, write to *prog*`.pp`. **Note:** This switch changes the order of allocation for uninitialized global variables. Programs which depend on global variables being allocated one after another must not use this switch.	2–64
`-e`	Issue warnings for language extensions.	2–64
`-err` [*file*]	**PC only.** Write error messages to *file*.	2–64
`-err+` [*file*]	**PC only.** Append error messages to *file*.	2–65
–h	Generate MC68881/MC68882 floating point instructions.	2–41
`-I` *dir1* [*dir2...*]	Define user `#include` directory(ies).	2–16
`-i`	Generate interleaved source and pseudo–assembly listing.	2–13

Option	Function	See:
`-ia`	Generate the object module by assembling compiler output. This is required if in–line assembly language is used.	2–13
`-id`	Suppress PC–relative addressing for data references. Use with `-pd` if separate code and data address spaces.	2–57
`-ih`	Assume routines called by interrupt handlers do not use floating point arithmetic (only matters with `-h`).	2–51
`-j`	Use short branches where possible.	2–52
`-k`	Use a single name space for structure fields.	2–65
`-ke`	On the PC, keep the compiler intermediate files. On the PC, also execute the phases of the compiler sequentially. On Unix hosts, execute the phases of the compiler sequentially, and keep the intermediate files (for technical support use).	2–65
`-L`	Define `int` as 4 bytes, `short` as 2 bytes. **NOTE**: `-L` is used by default when compiling for C++	2–23
`-l` [*lfn*]	Write output of listing options to file *lfn*. If *lfn* is omitted, write to *prog*`.lis`.	2–14

Option	Function	See:
`-M` [*depsfile*]	Generate a list of "make" dependencies which result from the various header files included during compilation. **Note:** An object module is not generated unless `-ca` is used.	2–65
`-m`	Use MC68881/MC68882 instructions for mathematical functions. This option has no effect unless the `-h` option is also supplied.	2–42
`-mp` [*protofile*]	Construct an ANSI C prototype declaration for each procedure defined in the compilation.	2–66
`-n5`	Do not reserve A5 for global data. **WARNING:** This switch is not compatible with the standard run–time library.	2–53
`-n6`	Allow the compiler to use A6 for other purposes. **WARNING:** This switch causes great problems for symbolic debuggers.	2–54
`-n7` [*n*]	Limit stack–fixup optimization to *n* bytes (0 if *n* is omitted).	2–55
`-na`	Turn off ANSI C language extensions.	2–68
`-nal`	Assume that the source contains no aliasing. **WARNING:** This switch is not safe for all programs.	2–33
`-nd`	Suppress detection of assignments to dead variables.	2–34

Option	Function	See:
`-nf`	Generate narrow–format interleaved source and pseudo–assembly listing. The narrow–format listing usually fits in 80 columns while the `-i` listing is `-p` and `-i` listings are 132 columns wide.	2–13
`-nh`	Suppress code hoisting.	2–34
`-nl`	Do not remove `LINK/UNLK` instructions. This guarantees that the stack is traceable by the debugger.	2–35
`-no`	Skip the optimizer.	2–35
`-np`	Stop the optimizer from putting more than one variable in a register.	2–36
`-nr`	Suppress the strength reduction optimization.	2–37
`-o` *ofn*	Write object module to file *ofn*.	2–68
`-opfile` *opts*	Supply command line options in a file *opts*.	2–68
`-os`	Optimize for space at the expense of time.	2–37
`-ot`	Optimize for time at the expense of space.	2–37
`-P` *"string*[=*value*]*"*	Predefine preprocessor variable *string*.	2–69
`-p`	Generate Wide–format pseudo–assembly listing.	2–13

Option	Function	See:
`-pack` *n*	Change alignment of data. `-pack 1` causes byte alignment in structures, even on word–like items. `-pack 2` causes word alignment, even on fullword–like items. `-pack 4` causes fullword alignment for fullword data.	2–24
`-pc`	Force position–independent forms for code (e.g., `BSR.L`).	2–58
`-pd`	Force position–independent forms for data.	2–58
`-ps`	Use short position–independent form for code (e.g., `BSR.W`). If `-pd` is also present, use short position–independent forms for data also. Requires total code (and, if `-pd`, data) less than 32K bytes for safe use.	2–59
`-pw`	Emit warnings for calls to undeclared functions.	2–69
`-q`	Generate real–assembly listing. Can be combined with `-i` for interleaved source.	2–13
`-S` *dir1* [*dir2...*]	Define system `#include` directory(ies).	2–16
`-s`	Generate source listing.	2–12
`-sc` *defclass* [*defclass2*]	Define default class(es) for **separate** data.	2–26
`-sd`	Treat all global data as **separate**.	2–27

Option	Function	See:
`-se`	**Unix only.** Run the compiler phases sequentially rather than as a Unix pipe (for technical support use).	2–69
`-si`	Allocate string literals in the `idata` segment. This causes the compiler to use A5–relative addressing for string literals.	2–61
`-sp`	Enforce strict ANSI C precision constraints. **Note:** This switch decreases the efficiency of generated code.	2–56
`-ss` *defseg* [*defseg2*]	Define default segment(s) for **separate** data.	2–27
`-V`	Very verbose mode. Identifies all driver actions as they are performed. This determines which phase was executing if the compiler aborts (technical support)	
`-v`	Verbose mode. Reports date, time, and status/result of compilation.	2–69
`-vv`	Assume all global variables are volatile, whether declared as volatile or not. **Note:** This is required for programs that do not use the ANSI volatile keyword appropriately.	2–37
`-w` [*n*]	Suppress warning messages of severity less than *n*.	2–70
`-x`	Generate cross–reference listing.	2–14

Note: Most of these options are also applicable to the C++ compiler. See the *C++ Compiler User's Guide* for more information

2.3 Usage

The InterTools C compiler translates C source programs into object modules containing machine language for the 68xxx family of microprocessors. Input to the compiler is one or more C source programs, which can "include" other files. The main output is one or more object modules suitable for linking with other modules. Object modules may also be catalogued in a library. Various kinds of listings may be generated to display the results of compilation.

Note: The compiler produces a machine–language object module, **not** an assembly–language program. The assembler is not run on the compiler output. The compiler can produce a listing that shows the assembly language equivalent of the C program: this is called the pseudo–assembly listing. The pseudo–assembly listing cannot actually be assembled without modification.

Note: The compiler normally produces a machine–language object module. It can also produce an assembly language output. The assembly language output can be assembled without modification.

The compiler will produce object code for the MC68000, MC68010, MC68020, MC68030, MC68040, MC68302, MC68332, MC68340, MC68360, MC68EC030, MC68EC040, or MC68060 instruction set, depending upon the manner in which it is invoked. The compiler will not emit instructions which do not exist on the specified target processor.

The compiler has a powerful optimizer phase to generate tight object code. This optimizer was designed to be used with embedded applications, and can be used safely even in the presence of memory mapped I/O and interrupt handlers.

The optimizer can be used with the InterTools source–level debugger, CrossView Pro, but it can make debugging considerably more complex. Use the `-do` switch to disable those optimizations which interfere with debugging. For more details about the interaction between the optimizer and the debugger, see the description of the optimizer switches.

See the *Using the Optimizer* section for a detailed description of the function and usage of the optimizer, as well as some general coding hints for getting better object code.

The compiler supports interrupt handlers in C. This feature is described in the *C Language Specifications* appendix.

The InterTools compiler fully supports the ANSI C standard. However, in the interest of backwards compatibility, some aspects of ANSI C are only supported in the presence of command line switches. For example, ANSI C requires the size of the `short` datatype to be at least 16 bits. By default the InterTools compiler maps the `short` datatype into an 8–bit integer. Therefore the InterTools compiler is not ANSI compliant unless `-L` or at least `-D s2s` is supplied.

Full ANSI C compatibility also requires the `-dd` switch and (under hardware floating point) the `-sp` switch. However, these switches have side effects which may affect code quality. See the relevant switch descriptions for more details.

Example

```
c68000 sieve.c
```

- Compile source in file `sieve.c`.
- Write object module to file `sieve.ol`.
- Search for `#include` files in the current working directory.
- No listings will be generated.

2.4 C Compiler Options: Detailed Descriptions

This section describes the C compiler options in more detail and provides examples of their use.

2.4.1 Listing Options

The listing options control the generation of the various listing files. Listings are not produced by default; the user must specify the appropriate options.

`-a, -s` Both `-a` and `-s` generate a source listing. The `-a` switch specifies that secondary `#include`'d lines be listed in addition to the primary source lines. If the `-l` switch is not specified, the source listing is written to file *prog*`.lis`.

`-i`

`-nf`

`-p`

`-q`

These switches control listings showing the generated code. There are two kinds of assembler listings. One is a "pseudo–assembly" listing. The pseudo–assembly listing contains opcodes, a location counter, and actual object code bytes. The pseudo–assembly listing resembles the listing produced by the 68k assembler; it cannot actually be assembled. If the `-l` switch is not specified, the pesudo–assembly listing is written to file *prog*`.psa`.

The other kind of assembler listing is a "real–assembly" listing. It can be assembled, and, if the `-ia` switch is present, it actually *is* assembled. It is written to the file *prog*`.s`. The real assembly listing resembles assembler source code. You cannot get a pseudo–assembly listing if the assembler is being used, that is, if the `-ia` switch is present, but you can ask the assembler to generate a listing. Use the `-ao` switch to pass listing options to the assembler.

Either of these listings can be interleaved with C source. The pseudo–assembly listing can be interleaved in wide or narrow format; use the narrow format listing to print on an 80 column terminal screen or printer, because it won't wrap to the next line.

Here are the possible combinations of options and what they do:

- For assembly listing without actually using the assembler:

  `-p` Wide–format pseudo–assembly.

`-i` Wide–format interleaved pseudo–assembly.

`-nf` Narrow format interleaved pseudo–assembly.

`-q` Real–assembly listing.

`-q -i` Interleaved real–assembly listing.

For assembly listings that use the assembler:

`-ia -q` Real–assembly listing (and assemble it).

`-ia -i` Interleaved, real–assembly listing (and assemble it).

`-l` [*lfn*] This switch controls the destination of listing output implied by other listing switches. If *lfn* is specified, the listing (if any) is written to file *lfn*. If *lfn* is omitted, the listing is written to file *prog*`.lis`.

If multiple listing options are selected, the results of all listing options will be put in file *lfn*. If multiple source files are given in one compiler invocation, *lfn* may not be specified. Instead, a separate listing file is generated for each input file. The listing output corresponding to *progx*`.c` appears in *progx*`.lis`.

`-x` Generate a cross–reference listing. If the `-l` switch is not specified, write the listing to *prog*`.xrf`.

Example

Compile multiple programs:

```
c68000 sieve.c subr.c -s -l
```

- Compile source in `sieve.c` and `subr.c`.
- Write object modules to files `sieve.o1` and `subr.o1`.
- Write source listings to files `sieve.lis` and `subr.lis`.

Example

Generate assembly and cross–reference listings in the same file:

```
c68000 hello.c -i -x -l
```

- Compile source in file `hello.c`.
- Write object module to file `hello.o1`.
- Write cross–reference and interleaved pseudo–assembly listing to file `hello.lis` (by using the combination `-i -x -l`).

Example

Generate source and cross–reference listings:

```
c68000 hello.c -a -x
```

- Compile source in file `hello.c`.
- Write object module to file `hello.o1`.
- Write source listing showing all `#include`'d files to file `hello.lis` (by using `-a`).
- Write cross–reference listing to file `hello.xrf` on the PC (by using `-x`) or `hello.xrf` on Unix Hosts (by using `-x`).

Source listing `hello.lis`:

```
1       #include "hello.h"
1   1   /* hello.h */
2   1   int i;          /* i gets declared here */

2       main()
3       {
4           printf ("Hello, world!\n");
5           i = 1;
6       }
```

Cross–reference listing `hello.xrf`:

```
i
    Def     :  hello.h          2
    Ref     :  hello.c      5

main
    Def     :  hello.c          2

printf
    Def     :  * undefined *
    Ref     :  hello.c          4
```

2.4.2 Include Options

`-I` *dir1* [*dir2* ...] Define one or more directories to be searched for user include files. The default is to search the directory containing the source file. No more than 32 user include directories may be specified.

`-S` *dir1* [*dir2* ...] Define one or more directories to be searched for system include files. The default is to search the current working directory. No more than 32 system include directories may be specified.

On the PC, the directory path in the `INCLUDE` environment variable is searched after the `-S` directories. (This environment variable may also be named `I2INCLUDE` to avoid conflicts with other software.)

Here are some `#include` directives as they might appear in a C program source file:

```
#include "file.h"
#include <file.h>
```

Note: `#include` without `" "` or `< >` is **invalid**. The first form is considered to be a user include; the second form is considered to be a system include. Nesting is limited to 10 levels of `#include` files on the PC and 15 levels on Unix.

When searching for user includes, the compiler first searches the directory containing the source file (default), then all `-I` directories in the order specified, followed by all `-S` directories. When searching for system includes, the compiler searches only the directories specified by the `-S` option, or the current working directory (default), if no `-S` is specified.

For both user include and system include files, the user can override the placement of the default directory in the search order by specifying an empty (null) directory name as an `#include` directory. An example of this appears below:

```
c68000 prog.c -I first -I -I third
```

In this case, the default directory (which is the directory containing the source file) is only searched at the position where the empty directory name appears, i.e., after the directory named `first` and before the directory named `third`.

Example

Specify other directories for `#include` files:

On the PC:

```
c68000 sieve.c -I smith\inc jones\inc williams\inc
```

On Unix hosts:

```
c68000 sieve.c -I smith/inc jones/inc williams/inc
```

- Compile source in file `sieve.c`.
- Write object module to file `sieve.o1`.
- The compiler searches for `#include` *<filename.h>* (system include) in the current working directory.
- The compiler searches for `#include` *"filename.h"* (user include) in the current working directory and the directories `smith\inc`, `jones\inc`, and `williams\inc` on the PC or `smith/inc`, `jones/inc`, and `williams/inc` on Unix hosts.

Example

Change the order of `#include` processing:

On the PC:

```
c68000 \usr\frank\sieve.c -S smith\inc jones\inc -S
```

On Unix hosts:

```
c68000 /usr/frank/sieve.c -S smith/inc jones/inc -S
```

- Compile source in file `\usr\frank\sieve.c` on the PC or `./usr/frank/sieve.c` on Unix hosts.
- Write object module to file `sieve.o1`.
- The compiler searches for files included as `#include` *<filename.h>* (system include) in the system include directories `\usr\frank,smith\inc,jones\inc,` on the PC or `/usr/frank` on Unix hosts and the current directory.
- The compiler searches for files included as `#include` *"filename.h"* (user include) in the source directory, `\usr\frank` on the PC or `/usr/frank` on Unix hosts, then in the system include directories named above, and finally in the current directory.

2.4.3 Datatype Options

The InterTools C compiler allows the user to redefine the size of the `int` `short` and `enum` datatypes to accommodate larger values, or to assure compatibility with other modules. You can specify whether the `char` datatype is to be treated as signed or unsigned.

In the case of enum types, the compiler supports four storage allocation strategies. By default, an enumeration type all of whose values are between –32768 and +32767 is stored in a signed 16–bit word, and other enumeration types are stored as a signed 32–bit word. The other strategies are:

1. Store an enumeration type whose values are between 0 and 255 as an unsigned byte; store other enumerations as a signed 16–bit word.

2. Store an enumeration type whose values are between –128 and 127 as a signed byte; store other enumerations in a signed 16–bit word.
3. Store all enumeration types as a 32–bit fullword.

The first two alternative strategies can result in more compact data. In general, the last alternative strategy results in bigger data, and perhaps, smaller and faster code. The last alternative strategy works best if the integer datatype is also redefined to be 32 bits long.

Note: It is essential that all modules which are linked together into one program be compiled with the same datatype options. This rule includes run–time library routines.

The characteristics of char, short and enumeration types do not affect the library. The compiler is distributed with run–time libraries that have been compiled both with and without the `-L` option.

For more information about the run–time libraries available, please refer to the *Compiler Library Organization* section in the *Linking Locator* chapter.

`-ab` Force word alignment for structures containing bit fields.

A bit field that is completely contained in a byte can be accessed via byte operations. Versions of the compiler prior to Release 8.0 sometimes used word operations to access some bit fields which were contained in a byte. This forced the compiler to give word alignment to all bit field structures, even those whose total size is one byte.

Later compiler releases use byte operations to access all bit fields that are completely contained in a byte. This means that word alignment is no longer necessary, and it is no longer enforced.

`-bb` Maintain backwards compatible bitfield storage layout.

This switch forces the compiler to use the same storage allocation algorithm as was used by InterTools C versions before version 7.1. The old algorithm disallowed bitfields bigger than 16 bits, and aligned a bitfield on the next halfword boundary if it would not otherwise fit completely in one halfword–aligned halfword. The newer algorithm allows bitfields up to 32 bits long, and aligns a bitfield on the next halfword boundary only if it would not otherwise fit completely in one halfword–aligned fullword. For example, a bitfield structure consisting of a 15–bit field, a 2–bit field, and a 15–bit field would be allocated in four bytes under the new strategy, and would require five bytes in the old strategy.

Note: The `-bb` switch is only for users who must maintain compatibility with the old storage allocation. It results in a less compact storage mapping.

`-D` *tbs* [*tbs*] Redefine built–in datatype.

The `-D` switch overrides the default size and sign of a built–in datatype.

Note: Use the `-D` switch carefully, since all modules intended to be linked together must be compiled with the same datatype length options. This rule includes run–time library modules.

Operands of the `-D` switch are triples of the form *tbs*, where *t* defines the datatype, *b* the number of bytes and *s* the signed/unsigned attribute. Legal values of *t*, *b*, and *s*, and their permitted combinations and defaults are:

t type	b	s
c char	1	s signed
e enum	2	u unsigned
i int	4	
s short		

Permitted Combinations of *tbs* and their defaults are:

Permitted:	c1s	c1u	e1s	e1u	e2s	e4s	i2s	i4s	s1s	s2s
Defaults:		c1u	e2s				i2s		s1s	

`-L` The `-L` switch is shorthand for the following combination of switches:

```
-D i4s s2s -P _LONGINT
```

The first option defines the `int` type as 4 bytes long, and the `short` type as 2 bytes long. The second option defines the preprocessor variable `_LONGINT`. You can use `_LONGINT` to define the length of an integer during compilation, allowing conditional compilation. Without `-L` `int` is 2 bytes and `short` is 1 byte. When compiling for C++, `-L` is the default.

Example

Select long integer option:

```
c68000 sieve.c -L
```

- Compile `int` variables as 4–byte signed data items; `short` variables as 2–byte signed.
- Write object module to file `sieve.ol`.

Note: `sieve.ol` must be linked with the long–integer runtime library.

`-pack` *n*

Change alignment of data. The value of *n* must be `1`, `2`, or `4`. The default mode depends on the processor. For the MC68020, MC68030, MC68040, MC68060 and the corresponding EC–series processors, `-pack 4` is the default. For other processors `-pack 2` is the default.

The easiest way to see the effect of the `-pack` switch is to consider this structure:

```
struct { char c; long l; };
```

Under `-pack 1`, this structure would be 5 bytes long. Under `-pack 2`, the compiler would ensure that the field `l` is word aligned. This requires a one–byte "hole" between `c` and `l`; thus the structure would be 6 bytes long. Under `-pack 4`, the compiler would ensure that the field "`l`" is fullword aligned. This requires a three–byte hole; thus the structure would be 8 bytes long.

The effects of alignment depend on the processor type. If a 68000–like processor attempts a word or fullword operation on an odd address, the processor will force an address exception. If a 68020–like processor attempts a fullword operand on an address which is not fullword aligned, then there is a performance degradation. The defaults (`-pack 2` or `-pack 4`) were chosen to maximize performance and avoid address exceptions.

Note that the `-pack 4` switch also ensures that all fullword variables are aligned on fullword boundaries, even if they occur outside structures. However, even under `-pack 1` individual word and fullword variables are word aligned. Thus `-pack 1` only affects structure layout.

Here are the two most common uses of the `-pack` switch:

- In a system which shares data between different processors, to ensure that the data is aligned to the maximum required by all the processors.
- To make data more compact at the cost of code speed.

WARNING: On a 68000–like processor, you should specify `-pack 1` only if you are prepared to handle the address exceptions that may result.

2.4.4 Separate Data Options

A brief overview of separate data options is given in this section. For more details, refer to the *Pragma Separate (Option Separate)* application note.

`-cs`
: Place all data declared as `const` into a separate segment `cdata` and class `constant`.

  Please see the *C Language Specifications* appendix for a description of the `const` type qualifier.

  This switch causes the compiler to segregate variables declared with the `const` attribute from other variables so they can more easily be allocated in ROM.

`-sc` *defclass* [*defclass2*]
: This option defines a default class or classes for separate data segments. When invoked with argument *defclass*, all separate segments whose class name is not otherwise specified have class *defclass*. When invoked with arguments *defclass* and *defclass2*, segments for initialized data have class *defclass* and segments for uninitialized data have class *defclass2*.

The `-sc` option may also be used with the `-ss` option to set defaults for both class and segment.

All modules which contain separate declarations naming the same segment with no class name must be compiled with the same `-sc` option. The linker will report an error if a segment is assigned different class names in different modules. Improper use of the `-sc` switch can cause such errors.

`-sd` Treat all global data as separate. This option has the same effect as a `#pragma sep_on` directive on the first line of the source file being compiled. For a more detailed explanation of this feature, please read the *Pragma Separate (Option Separate)* application note.

`-ss` *defseg* [*defseg2*] This option defines a default segment or segments for separate data. When invoked with argument *defseg*, all separate data whose segment name is not otherwise specified are allocated in segment *defseg*. When invoked with arguments *defseg* and *defseg2*, initialized separate data whose segment is not otherwise specified are allocated in segment *defseg* and uninitialized separate data is allocated in segment *defseg2*.

The `-ss` option may also be used with the `-sc` option to set defaults for both class and segment.

The linker will report an error if a segment is assigned different class names in different modules. For this reason, the `-ss` segment assignments are NOT applied to separate variables whose class name is specified, unless the specified class name is the same as the "expected" class name for the `-ss` segment. If the `-sc` option is present, the expected class name is the `-sc` name. Otherwise the expected class name is "separate".

All modules compiled with the same `-ss` option should supply the same `-sc` option to avoid link–time errors.

Note: `#option` is equivalent to `#pragma`.

2.4.5 Optimizer Options

There are several switches to control the behavior of the optimizer. There are two main reasons why one might wish to do this. The first is to allow the optimizer to perform optimizations which are not safe in general but which *are* safe for the particular module being compiled. The second is to make the object code easier to understand or debug.

The InterTools source-level debugger, CrossView Pro, can be used with optimized code. However, certain unexpected behavior can be caused by the transformations performed by the optimizer. These anomalies are described briefly below with the related compiler switches.

`-ai`

Perform automatic inline procedure expansion.

The compiler supports a limited form of automatic inline procedure expansion. You can request this optimization by specifying the `-ai` (automatic inline) command-line switch.

A procedure is said to be "expanded inline" if a copy of the body of the subroutine is inserted in place of the usual call operation. Generally speaking, inline procedure expansion represents a trade-off of code space for execution speed. You save the execution time spent in a call and return, but the compiler must generate a whole new copy of the called subroutine body at every inline call.

There are limits on the kinds of routines which can be expanded inline; inline expansion is currently limited to two cases:

- static subroutines which return a non-aggregate value and which contain no flow-of-control statements (that is, no

loops or "`if`" statements are allowed; however, conditional "`? :`" expressions are allowed.)

- static subroutines of type `void` (that is, which return no value)

There is also a limit on the overall size of an inline routine, and a limit of six arguments in an inline routine.

When the `-ai` switch is enabled, the compiler behaves as follows:

- When processing a static subroutine, the compiler creates an internal copy of the body of the subroutine. If the routine turns out to be unsuitable for inline expansion, the compiler emits the body as usual. Otherwise, the compiler emits no code for the procedure, and retains the internal copy of the body for future use.
- When the compiler processes a procedure call to a routine for which it has retained a copy, it duplicates the body in place of the call.
- At the end of the compilation unit, the compiler checks if there is any need to emit an out-of-line body.

  There are two reasons the compiler might need to emit an out-of-line body: first, there may have been calls which the compiler did not expand inline because they appeared before

the definition of the routine; second, the compiler might have made some use of the address of the function (for example, the compiler might have assigned the address to a pointer-to-function).

If neither of these conditions apply, the saved body is discarded.

To take advantage of inline procedure expansion across compilation units, you may be tempted to place the procedures which you intend to expand inline into a set of ".h" include files, together with their bodies. This would allow these procedures to be expanded inline in many different compilation units.

However, we must warn you that code structured in this manner cannot be debugged effectively with or without `-ai`, even though it will compile and run properly. The problem is that most debug symbol table formats, including the `.abs` file used by CrossView Pro, cannot describe programs whose code comes from more than one source file. In fact, the formatter treats the first file with line number marks in it as the "primary" source file, and discards marks from the other files.

This restriction will be removed in later releases of the compiler and debugger; however, until then, we cannot recommend the methodology of putting source in include files.

`-do` Disable optimization which substantially interferes with debugging. This is equivalent to `-nd`, `-nh`, `-nl` `-np` and `-nr`.

Note: Use of the `-do` switch may make the generated code significantly larger and slower.

`-nal` Assume that the source contains no aliasing.

WARNING: This switch is not safe for all programs.

To understand aliasing, consider the following piece of C code:

```
int i;
func() {
    int *pi;

    pi = &i;
    i = 0;
    *pi = 7;
    if(i == 0)
        printf("test failed\n");
    }
```

This code contains an example of *aliasing*. This means that the variable `i` is referred to both by name, in the assignment of 0 to `i`, and through the pointer `pi`, in the assignment of 7 to `*pi`. Aliasing refers to the use of a pointer to an object and that same named object in the *same* function.

Because of the possibility of aliasing, when an assignment is made through a pointer variable, the optimizer must forget its knowledge of all variables that could be pointed to by that pointer. This includes all extern variables, and any locals that ever have their addresses taken in the current function. The example above shows what could happen if the optimizer did not take aliasing into account. If the optimizer

did not know that `i` could be modified via the pointer `pi`, then it might decide that `i` was still equal to zero, and thus incorrectly decide that the `if` condition must be true.

`-nd`

Suppress detection and removal of assignments to dead variables.

Normally the optimizer will remove stores into local variables which are not subsequently referenced. This means that reading a variable in the debugger will not always yield the value last assigned to it.

Note: The `-nd` switch may make the generated code somewhat larger and slower.

`-nh`

Suppress code hoisting.

The code hoister is a part of the optimizer which attempts to hoist instructions out of loops. After code hoisting, part or all of a statement within a loop may be performed outside the loop. Normally when the debugger stops at a C statement one can assume that no instructions from that statement have been executed, but this is no longer true after code hoisting.

Note: The `-nh` switch may make the generated code run significantly more slowly.

`-nl` Do not remove `LINK` and `UNLK` instructions.

The `LINK` instruction is used in procedure prologue to establish a stack frame. Many procedures do not require a stack frame, since the optimizer can often pack all local variables into registers. However, when `LINK` instructions are removed the debugger cannot find all the active procedures by stack traceback.

For example, suppose procedure `f1` calls procedure `f2` which calls procedure `f3` which calls procedure `f4`, and that the `LINK` instruction in procedure `f3` was optimized away. The debugger's stack traceback analysis starting from procedure `f4` would incorrectly conclude that the procedure `f3` was called from procedure `f1` rather than from procedure `f2`.

Note: The `-nl` switch may make the generated code run more slowly.

`-no` Skip the optimizer.

This switch bypasses the optimizer phase of the compiler.

`-np`

Stop the optimizer from putting more than one variable in a register.

By default the optimizer attempts to put as many different variables into the same register as possible. Of course, two variables may occupy the same register only if their lifetimes do not overlap.

If two different variables are allocated in the same register, then reading or writing those variables in the debugger can have unexpected results. For example, assigning a variable before its first use can cause a different variable to be corrupted. Similarly, reading a variable after its last use can deliver an incorrect value.

This switch directs the optimizer not to pack more than one variable to a single register.

Note: The `-np` switch may make the generated code significantly larger and slower.

`-nr` Suppress strength reduction. Strength reduction is an optimization that typically turns multiplies into additions inside a loop.

In the classic case, the loop:

```
int a[10];
for (i=0; i <10; i++) {
   a[i] = 0;
}
```

can be turned into the equivalent of:

```
int a[10],*pnt;
for (pnt = &a[0]; pnt<&a[10];) {
   *pnt++ = 0;
}
```

which is both smaller and faster.

`-os` Optimize for space. Choose smaller but slower code sequences.

`-ot` Optimize for time. Choose faster, but larger code sequences.

Note: In the absence of either the `-os` or `-ot` switch, the compiler optimizes for time over space.

`-vv` Assume all global variables are volatile.

Most variables in a C program have the characteristic that if they are referenced twice with no intervening stores, either direct stores or through pointers, then both references will deliver the same value. A variable is called

"volatile" if it does not have this property. Generally the only way a variable can be volatile is if that variable is located over a memory–mapped I/O port, or if it could be modified by an asynchronous interrupt handler.

It is critical the that optimizer know when it is dealing with a volatile variable, since optimization can cause programs which use volatile variables to execute incorrectly. Here is a simple example:

```
extern int interrupt_happened;
void wait_for_interrupt() {
   interrupt_happened = 0;
   while (interrupt_happened ==
0);
}
```

This programs expects an external interrupt handler to change the global variable `interrupt_happened`. However, the optimizer will, by default, assume that `interrupt_happened` is always zero, because it can see no code that can affect this variable between its assignment and the test within the loop. Thus it would compile the "while" loop into a jump–to–self instruction.

The best way to avoid problems like this is to use the ANSI volatile keyword. That is, the declaration should be:

extern volatile int interrupt_happened;

However, if you do not feel confident that you can locate and appropriate qualify all your

volatile variables, then you can still avoid inappropriate optimizations by using the `-vv` switch. This tells the compiler to treat **all** global variables as volatile.

Note: The `-vv` switch may make the generated code significantly larger and slower.

2.4.6 Floating Point Options

`-68` Software floating–point compatibility mode.

This option is intended to facilitate the migration of a system using software floating point to one using hardware floating point. It directs the compiler to use the software floating point linkage conventions even though the hardware floats option is selected. This option has no effect unless the hardware floats option, `-h` is also selected.

By default, function return values of type `float` are passed in register FP0 if the hardware floating point option is selected, and in register D0 otherwise. Function return values of type `double` are passed in FP0 in the hardware floating point case, and in memory otherwise. (See the *Compiler Run Time Conventions* appendix for more details.) This option directs the compiler not to return float/double function values in FP0 even when the hardware floating point option is selected.

Suppose, for example, that you have a collection of assembly language routines which return float or double values and which are called from C code. Suppose further that these routines were coded using the software floating point linkage conventions, as described in the paragraph above. If you re–compile the calling C routine(s) with the hardware floats option, then the assembly language routines will not operate correctly. However, if you also supply the `-68` option, then the assembly language program can be used as is. This will allow you to modify the assembly language routines gradually, and then you can switch over to the more efficient hardware floating point linkage conventions when you are ready.

Note: Use the `-68` switch carefully, since all modules intended to be linked together must be compiled with the same linkage conventions.

Care must be taken with any run–time library functions which return `float/double` values. Most mathematical functions (e.g., `sin`) can be expanded in–line by using the `-m` switch. If other such library routines are needed, e.g. pow, then they should be re–compiled with `-68`. Refer to the `-m` switch for related information concerning mathematical functions.

`-h` Generate hardware floating point instructions. Except for the MC68040 and the MC68060, the default is to use software floating point. For the MC68040 target, hardware floating point instructions are generated by default.

Note: After compiling with the `-h` switch, you must link the object module with the hardware floating point library.

`-m`

Make calls to transcendental functions using floating point instructions. The option is effective only when the hardware floating point option, `-h`, is also selected, or if the target is the MC68040 or MC68060.

This option directs the compiler to assume that calls to functions named `sin`, `sqrt`, and so on actually are invocations of the corresponding mathematical functions, and *not* user–defined subroutines. Furthermore, it doesn't generate code to set the global variable `errno`. This is non–ANSI behavior because the ANSI standard requires `errno` to be set if arguments are out of bounds.

For example, if the argument of `sqrt` is less than zero, then the ANSI standard requires that the `sqrt` library routine set `errno` to `EDOM`, a constant defined in the `errno.h` include file. Under the `-m` switch, the only code generated for a call to `sqrt` would be a `FSQRT` instruction. Without `-m`, the generated code would be a call to a library routine that would range check the argument, set `errno` if it is out of bounds, and then use `FSQRT` to compute the result.

Programs that do not check the `errno` variable after calls to mathematical library routines will produce the same results when compiled under the `-m` option.

Note: The `math.h` system include file supplied with the run–time library must be `#include`'d in the C source to supply external declarations of the mathematical functions.

Note: Even if the `-m` switch is not supplied, calls to mathematical functions that have no range restrictions of their arguments are expanded in–line by default. The routines treated in this way are `atan`, `cos`, `cosh`, `exp`, `fabs`, `sin`, `sinh`, `tan`, and `tanh`. In this case the full ANSI C semantics are preserved. If in–line expansion is **not** desired, it can be avoided by `#undefing` the function after the `math.h` include file is included. For example, this would force real out–of–line calls to `cos`:

```
#include "math.h"
#undef cos
```

The following table summarizes which routines are expanded in–line by default. On the MC68040 and the MC68060, the only floating functions are `sqrt` and `fabs`; all other routines are done out of line.

Subroutine	Mathematical Function
`acos`	arc cosine
`asin`	arc sine
`*atan`	arc tangent
`*atanh`	hyperbolic arc tangent
`*cos`	cosine
`*cosh`	hyperbolic cosine
`*exp`	exponential
`+*fabs`	absolute value
`log`	natural logarithm
`log10`	base 10 logarithm
`log2`	base 2 logarithm
`*sin`	sine
`*sinh`	hyperbolic sine
`+sqrt`	square root
`*tan`	tangent
`*tanh`	hyperbolic tangent

* Expanded in–line by default.

\+ Expanded in–line on MC68040 and MC68060.

2.4.7 Code Generation Options

`-aa` Align each procedure on a 16–byte boundary.

`-ac` Align only the first procedure on a 16–byte boundary.

These two switches (`-aa` and `-ac`) are intended for use with the 68040. They help optimize the instruction cache by aligning subroutines on a 16–byte "line" boundary. `-aa` aligns every subroutine; `-ac` aligns only the first subroutine in the compilation unit.

The 68040 loads its internal instruction cache in units of 16 bytes called "lines". Lines are always loaded from 16–byte aligned boundaries. To see how this affects program execution, consider the case of two procedures: f1 and f2. Suppose f1 is located at address 1600, a line–aligned address, and that f2 is located at address 3208, eight bytes after a line–aligned address. Suppose further that neither is resident in the cache.

When f1 is entered, the 68040 will load the line containing the entry point for f1 (addresses 1600–1615) into the cache. This loads 16 bytes of the procedure f1. The 68040 will not need to load more code into the cache until it executes the first 16 bytes of f1. Now consider what happens when f2 is entered. Again, the 68040 will fetch the line containing the entry point for f2 (addresses 3200–3215). This will only obtain 8 bytes of the procedure f2. This means that the 68040 may need to load the instruction cache sooner, resulting in a delay.

The `-ac` switch is appropriate when the compilation unit consists of a relatively small

package of procedures that often call one another. Thus when you enter the package you expect to see the whole package get loaded into the cache. The `-aa` switch is appropriate when the compilation unit consists of a package of routines that do not call one another. Thus when you enter the package you want to load the minimum amount into the cache.

Aligning all procedures on a line boundary is probably not a good idea, because it does make the total code larger (on the average, 8 bytes per procedure). This makes it harder to cover the code with a limited size cache. However, it makes good sense to align the most frequently executed procedures or groups of procedures.

`-ar` Use alternate register usage conventions.

Under the `-ar` switch, the compiler uses a different set of register conventions which more closely match that used by other 68000–family compilers. Under the normal conventions, registers D0, D1, A0, A4, FP0, and FP4 are considered scratch registers, and pointer return values come back in A0. Under the alternate conventions, registers D0, D1, A0, A1, FP0, and FP1 are scratch registers, and pointer return values come back in D0 (the same as integers).

The `-ar` switch makes it easier to use the InterTools compiler with assembly language which was designed for use with another compiler.

Of course, all modules in a program must be compiled using the same register conventions. This rule also applies to the run–time library. Therefore it is necessary to recompile the run–time library using `-ar` in order to make a library which can be used with `-ar` compilations. See the *Run–time Library* chapter in the *InterTools Reference Manual* for more details on how to rebuild the run–time library.

`-b5` Use 32–bit A5–relative offsets.

WARNING: This switch may make the generated code significantly larger and slower.

The compiler addresses non–separate data via the A5 register. Normally the compiler imposes a 64K byte limit on A5–relative data, which

allows the compiler to assume that A5–relative offsets fit in 16 bits.

The -b5 switch removes the 64K byte limitation on A5–relative data. The compiler must then use 32–bit A5–relative offsets. This means it can no longer use the efficient "A5 plus 16–bit displacement" addressing mode. Instead it must use the "A5 plus 32–bit displacement" mode. On a 68000–like processor this more complex addressing mode is not available, so a sequence of instructions are necessary instead.
For example:

```
extern long i;
i = 1;
```

Default code:

```
MOVE.L  #1,_i-data(A5)
```

Code under -b5, 68000 target:

```
MOVE.L  #_i-data,D0
MOVE.L  #1,(A5,D0.L)
```

Code under -b5, 68020 target:

```
MOVE.L  #1,(_i-data,A5)
```

-C

Old run–time model compatibility mode.

This option is intended to facilitate the migration of a system built with a version of the InterTools compiler earlier than 7.0.

The run–time model for the 68000 family compiler was changed slightly, starting with the version of the compiler which supports the optimizer (Version 7.0 or later). The change affects the compiler's run–time convention with respect to procedure calls and "preserved registers."

The newer run–time model requires that register D2 through D7 be preserved by any procedure called from C, while the old run–time model only required that D2 through D4 be preserved. In the hardware floating point case, registers FP1 through FP73 and FP5 through FP7 must be preserved where before only FP1 through FP3 needed to be preserved. This has two implications:

Assembly language programs which are called from C code may have to be modified to save and restore these additional registers in their entry/exit code if they were coded assuming the old run–time model.

C code compiled under the older run–time model may not be called by code compiled under the newer run–time model. The converse, however is **not** true: code compiled under the newer run–time model may be called from code compiled under the older run–time model. In

particular, the newer model run–time library is compatible with older compilers, and with code compiled under the `-C` option.

The run–time model was changed to allow the compiler to use more registers for register variables. It has a very substantial impact on code quality, especially with the optimizer turned on. We recommend that the `-C` switch be used only as a

stopgap measure until any affected assembly language routines have been modified.

`-cc` *classname* Set class of generated code segment to *classname*.

The compiler generates one segment for each source model to contain the generated machine instructions. By default this segment is associated with the class name "`{code}`". This option chooses a different class name.

It may be convenient to compile collections of related modules with this option. The code can be distinguished in gsmap listings, and the code from all these compilations can be forced into a single address range with a single `locate` command. See the *Linking Locator* chapter for more details.

`-ih`

Assume routines called by interrupt handlers do not use floating point arithmetic. Only matters with `-h`.

An interrupt routine that uses floating point registers must preserve the state of the floating point unit. This requires several instructions, in the entry/exit sequence starting with an `FSAVE`. Normally any interrupt routine which makes a subroutine call must do the same. This is necessary because the compiler fears that the called routine may do floating point arithmetic. However, if you know that the routines called by your interrupt handlers do not do any floating point arithmetic, then these saves are unnecessary. They can also be quite slow.

By supplying the `-ih` switch, you can tell the compiler not to worry about the routines called from your interrupt handlers. Of course, if the interrupt handler itself uses floating point registers, then they will be saved on entry nevertheless.

`-j`

Use short branch instructions where possible. This switch may decrease code size, depending upon the nature of the source program.

When the compiler needs to emit a forward branch, there are two alternative strategies it could use: emit a short or long branch instruction. If the target of the branch instruction is reachable by a short branch, then the first strategy will produce smaller object code. If not, then the second strategy will produce smaller object code. The only way to know which strategy is better for any particular input program is to try both options and pick the winner. The default is to emit only long branches.

`-n5` Do not reserve A5 for global data.

By default, the compiler uses A5 to address non–separate global and static data. However, if you have no such data, then the A5 register is effectively unused. In that case, the `-n5` switch allows the compiler to use one more register. This can result in significant code improvement in subroutines that use a lot of pointer variables.

WARNING: This switch is not compatible with the standard run–time library, because the library itself has some global and static data. For example, the variable `errno` is defined by the library. In order to use the `-n5` switch, you must recompile the library using a compilation switch such as `-ss`, `-sc`, or `-sd`, which makes all data separate. See the *Building Libraries That Do Not Use A5* application note for a detailed explanation of how to do this.

–n6

Allow the compiler to use A6 for other purposes.

WARNING: This switch causes great problems for symbolic debuggers.

The `-n6` switch directs the compiler to use the A7 register (the stack pointer) to access variables on the stack. This has the advantage of freeing up an additional register for use by the code generator, which can make a big difference in subroutines which use lots of pointer variables. Code compiled with `-n6` may successfully be mixed with code compiled without `-n6`. In particular, it is NOT necessary to rebuild the run–time library with `-n6`.

The main disadvantage of the `-n6` switch is that the code generated under `-n6` is more difficult to read and debug. The CrossView Pro source level debugger is completely unable to locate stack variables or trace the stack when debugging code compiled with `-n6`. The main problem is that the A7 register changes frequently, so it is not easy for a debugger to calculate the A7 offset of a variable at any given point in a program. The A6 offset, in contrast, is always constant and easy to manipulate.

`-n7 [n]`

Limit stack–fixup deferral to n bytes (0 if *n* omitted).

The `-n7` switch is used to suppress or limit stack–fixup optimization. To explain why this might be necessary, we must describe how this optimization works.

After a procedure is called, the caller must pop the parameters off the stack. This operation is called a "stack fixup". The compiler attempts to optimize stack fixups by delaying them as long as possible. This may allow the compiler to do several fixups in one operation. For example,

```
f1(1,2,3);
f2(4,5,6);
f3(7,8,9);
```

By default the compiler would generate only one fixup, after the last call.

The problem with this optimization is that it may greatly increase the total amount of stack space required by the application. In this example, the parameters passed to `f1` and `f2` would still be on the stack when `f3` is called. If this construct appeared in a highly recursive procedure, then that extra space on each activation could become quite large. In this case, the `-n7` switch can be used to limit stack fixup delay to n bytes. Just "`-n7`" alone or "`-n7 0`" disables the stack fixup optimization entirely.

`-sp`

This switch requires the compiler to obey strict ANSI C rules for floating point precision at the cost of code efficiency. ANSI C permits floating point expressions to be evaluated in greater precision than their type, but it does not permit variables to be stored in greater precision than their type. Allowing the compiler to store variables in greater precision than their type is very important, because it means that variables of type `float` or `double` can be allocated in floating point registers. This is an enormous improvement in efficiency over allocating such variables in memory, but it does mean that these variables essentially take on extended precision.

Generally speaking, greater precision is desirable. Since the precision of floating point expressions is indeterminate, strictly controlling the precision of variables has little practical benefit. However, it is true that the exact value of a floating point result is less predictable when the precision of variables is indeterminate. This switch prevents user variables of type `float` or `double` from being allocated into floating point registers.

2.4.8 Position–independent Code Options

Position independence is a very complex issue. For more details, see the *Position–independent Code* application note.

`-id` Suppress PC–relative addressing for data references.

This switch is ignored unless `-pd` is also present. It is intended for use in systems where program–space fetches to data will not work properly, as is the case where code and data reside in different address spaces. Consider these two code sequences:

```
MOVE x(PC),D0
```

and

```
LEA  x(PC),A0
MOVE (A0),D0
```

The first sequence loads a word from program space. The second sequence loads a word from the same address, but it performs a data fetch, not a program fetch. The second sequence would be substituted for the first under the `-id` switch.

`-pc`

Force position–independent forms for code, e.g., `BSR.L`.

Uses PC–relative addressing to achieve position independence. Here are some examples:

```
extern void f();
extern void (*p)();
f();
p = f;
```

Default code:

```
JSR     _f
MOVE.L  #_f,_p-data(A5)
```

Code under `-pc`:

```
BSR.L      _f
LEA      (_f,PC),A0
MOVE.L  A0,_p-data(A5)
```

-pd

Force position–independent forms for data.

This only affects separate data and string literals. Here are some examples:

```
char *q;
#pragma separate p
char *p;
p = "abc";
q = p;
```

Default code:

```
MOVE.L  #__N1,_p
MOVE.L  _p,_q-data(A5)
```

Code under -pd:

```
LEA     (__N1,PC),A0
LEA     (_p,PC),A4
MOVE.L  A0,(A4)
MOVE.L  (_p,PC),_q-data(A5)
```

-ps

Use short position–independent form for code, e.g., BSR.W. If -pd also present, use short position–independent forms for data also.

WARNING: The total size of code must be less than 32K bytes for safe use of this switch. If -pd is also present, then total code plus data must be less than 32K bytes.

This switch tells the compiler that it may assume that all PC–relative offsets will safely fit

in 16 bits. If they do not actually fit, then use of this switch will cause an bounds error in the link or format stage of processing.

This switch significantly improves efficiency, especially on the 68000, 68010, and 68302 processors. It allows the compiler to choose the efficient "PC+16–bit displacement" addressing mode rather than the "PC+32–bit displacement" mode. On a 68000–like processor this more complex addressing mode is not available, and so a sequence of instructions are necessary instead. For example:

```
#pragma separate x
long x;
void f();
_f(x);
```

Code under `-pc -pd`, 68000 target:

```
MOVEA.L #_f-*-8,A0
LEA     (PC,A0),A0
MOVE.L  (A0),-A7
MOVEA.L #_f-*-8,A0
LEA     (PC,A0),A0
JSR     (A0)
```

Code under `-pc -pd`, 68020 target:

```
MOVE.L  (_x,PC),-(A7)
BSR.L     _f
```

Code under `-ps -pd`:

```
MOVE.L  _x(PC),-(A7)
BSR     _f
```

`-si`

Allocate string literals in the idata segment. This causes the compiler to use A5–relative addressing for string literals.

There are two strategies available for achieving position–independence for data. One is to use the `-pd` switch to cause PC–relative addressing for string literals and separate data. However, this requires that the string literals and separate data be moved as a rigid unit with the code. This may not be possible under some environments.

The other strategy is to make all non–stack data A5–relative. This requires avoiding the options and pragmas that cause separate data to be generated. Then all that remains is to make the string literals A5–relative. The `-si` switch does this.

2.4.9 Miscellaneous Options

`-0` (Zero) Display the version number of executables (for technical support use).

`-ao` *options* Pass specified options to the assembler step.

Options must be in quotes if there's more than one given. Use only with `-ia`.

This option is used to specify additional options to the assembler when assembling compiler output. It can be used, for example, to specify a macro library that defines macros for use within in–line assembly language insertions. See the *Assembler* chapter for more details.

`-ca` Continue compilation to completion even if the `-E` or `-M` switches are specified.

`-d` Include symbolic debug information in the object module. The default is no symbolic debugging information.

The linking locator and formatter programs pass symbol information through to their output files. Eventually the symbol information will reside in a hex output file, or in a debugger symbol file to be read by CrossView Pro. Symbol information can also be displayed with the symbol list utility.

`-dd` Allow ANSI–style duplicate declarations.

The ANSI C standard permits certain kinds of multiple definitions. For example, the following program is legal ANSI C:

```
int i;
int i = 1;
```

By default the InterTools compiler gives an error if more than one definition for a variable is present in a single module. Put another way, the `extern` keyword would be required on the first declaration. The effect of the `-dd` switch is to permit multiple definitions in a single module, as required by ANSI C.

Note that multiple definitions placed in separate modules are still illegal. The two lines above, if compiled separately would cause an error at link time. This treatment, called the "def–ref" model, is allowed by the ANSI standard and is the most common among modern C compilers.

When the `-dd` switch is present, the compiler delays emitting data allocations for uninitialized variables until the end of the compilation unit. (It does this in case an initialized definition turns up later.) This delay affects the order of allocation of uninitialized variables in memory.

It is expressly illegal for a C program to rely on the order of storage allocation, but some programs do. Also, some programs have latent bugs that only become apparent when the global variables are reordered. Programs which execute differently with and without the `-dd` switch should be examined for constructs which assume that variables are allocated one after another in the `udata` area.

Note: Programs which depend on the order of storage allocation must not use the `-dd` switch. However, the `-dd` switch is necessary for full ANSI compliance.

`-E` [*pfn*]

Generate a listing of preprocessor output. If *pfn* is omitted, the listing is written to file *prog*`.pp`. The default is no preprocessor listing. Compilation is halted after the preprocessor step (and no object module is generated), unless the `-ca` switch is also specified.

If `-E` is specified and `-ca` is not also specified, then the compilation is stopped after the preprocessor stage so that no object module is produced.

Note: If `-E` is supplied and `-ca` is not also supplied, then the compilation is stopped after the preprocessor stage, so no object module is produced. This allows the preprocessor to be run on source files that do not contain legal C code, such as C++ source.

`-e`

Issue a warning for use of language extensions. The C code containing the extension(s) is processed, but the warning alerts the user to the use of the non–standard feature. This warning is not affected by the `-w` switch.

`-err` [*file*]

PC only. Write error messages to file *file*. If *file* does not exist, it will be created. If *file* does exist, it will be overwritten. If *file* is omitted, then error output will be redirected to standard output.

`-err+` [*file*]
: **PC only.** Just like `-err`, except output will be appended if *file* exists.

`-k`
: Use a single name space for all structure fields. The default is to use separate name spaces for each structure type. When `-k` is specified, the compiler flags as an error the occurrence of the same field name in different structures.

`-ke`
: Execute the phases of the compiler sequentially, and keep the intermediate files. For technical support use.

`-M` [*depsfile*]
: Generate a list of "make" dependencies which result from the various header files included during compilation. Write the listing to *depsfile*. If *depsfile* is not specified, the listing output corresponding to *progx*`.c` appears in *progx*`.lis`.

  If `-M` is specified and `-ca` is not also specified, the compilation is stopped after the preprocessor stage so that no object module is produced.

  The compilation is halted after the preprocessing step (and no object module is generated), unless the `-ca` switch is also supplied.

Note: If `-M` is supplied and `-ca` is not also supplied, then the compilation is stopped after the preprocessor stage, so no object module is produced.

Note: The compiler will not write more than one listing for a source program to the default listing file, prog.lis. If you invoke the compiler using the -M switch without depsfile and try to generate other listings without using the -l switch (to redirect output), the compiler will only write a listing of "make" dependencies to prog.lis. Other listing information will not appear in any new file.

-mp [*protofile*] Construct ANSI–style prototype declarations for each procedure.

A function prototype is a kind of procedure declaration which indicates the types of the arguments expected by a procedure. Function prototypes are probably the single most useful new feature in ANSI C, because they allow the compiler to prevent the common error that results when the type of an actual parameter does not match that expected by the called procedure. Function prototypes are described in some detail in the *C Language Specifications* appendix.

The –mp switch is provided to help users who have non–ANSI C code to take advantage of function prototypes. The –mp switch causes the compiler to generate a prototype declaration for each of the procedures defined in this compilation. You may want to edit the generated "header" file (named prog.ah if protofile is absent) to add comments and so on.

After you have constructed prototype header files for all the procedures in your system, you

should add `#include` directives so that each module has access to prototype declarations for all the procedures that it calls. This ensures that all calls can be checked for argument mismatches and the appropriate conversions can be automatically generated by the compiler.

In C, function declarations come in two forms: "old style" and "new style". The "old style" is the K&R C syntax, e.g.,

```
f (x, y)
        float x;
        short y;
{
        ...
```

The "new style" is the ANSI C prototype syntax, e.g.,

```
g (double x, int y)
{
        ...
```

Under the old K&R rules, an outgoing parameter of type float was converted to double, and outgoing parameters of integral types smaller than int were converted to int. Therefore the function `f` above really expects a `double` and an `int`, not a `float` and a `short`, as it appears. Therefore the prototype generated under `-mp` would look similar for `f` and `g` above,

```
f (double, int);
```

This is NOT the same as would be generated for this procedure:

```
h (float x, short y)
{
        ...
```

The prototype generated for this procedure would look like this:

```
h (float, short);
```

`-na` Disable ANSI C language extensions, including new keywords. Using the `-na` switch turns off all ANSI C additions that would make legal, pre–ANSI C source code compile incorrectly.

`-o` *ofn* Write the object module to file *ofn*. The default is to write to file *prog*`.o1`.

`-opfile` *opts* This switch causes the compiler to read command line options from file *opts*.

`-P` *"string[=value]"* This switch has the same effect as one of the following C statements, depending upon whether *value* is specified.

```
#define string
#define string value
```

`-pw` Emit warnings for calls to undeclared functions.

In C, undeclared identifiers are implicitly declared as "external `int` function". This rule can mask real errors, producing code that will not execute properly. This is especially true in programs that rely on function prototypes to coerce arguments to the correct type.

This option causes the compiler to emit a warning message when it generates an implicit declaration. If all non–prototype ("old style") function declarations are removed, this switch effectively guarantees that a prototype will be in force at every call.

`-se` **Unix only.** Run the compiler phases sequentially rather than as a Unix pipe.

`-V` Very verbose mode. Reports date, time, and status/result of compilation.

`-v` Verbose mode. Identifies compiler phases as they are invoked. This helps determine which compiler phase was executing if the compiler aborts (for technical support use).

`-w [n]` Suppress warning messages of severity less than *n*.

The compiler generates warnings for non–portable or non–standard uses of the C language. Warning severities vary from 1 to 10 (1=least severe to 10=most severe), depending upon the error. If omitted, *n* defaults to 11, i.e., all warnings are suppressed. Warning severities are listed in the next section. The default is to issue all warning messages.

Example

Include debugging information:

```
c68000 sieve.c -d
```

- Write object module to file `sieve.ol`.
- Include symbol table information in the object module. This information can later be used by the symbolic debugger.

Example

Construct a header file of ANSI C prototype declarations:

```
c68000 prog.c -mp
```

- Write object module to file `prog.ol`.
- Write a header file, `prog.ah`.

Assume that `prog.c` contains the following lines:

```
int   f(x,y)
   double x;
   short y;
{ ...}
```

The resulting header file, `prog.ah`, would be:

```
int   f(double,

      int);
```

2.5 Using the Optimizer

Generally speaking, the optimizer is completely automatic. No source changes are necessary to use the optimizer, and optimized and non–optimized modules can be freely mixed.

The optimizer must build an intermediate representation of an entire procedure in order to perform its functions. If the optimizer does run out of space, these modules may be compiled without the optimizer by using the `-no` switch.

Optimization may make the generated code harder to debug. For example, the optimizer may hoist code out of a loop, making it impossible to set breakpoints within that code. When debugging is anticipated, the `-do` switch can be used to disable those optimizations which interfere with debugging. This is preferable to `-no` which disables many more optimizations.

The optimizer uses many different techniques to improve the quality of the generated code. The main techniques are described in the *Optimizations Performed* section below. By far the most significant optimization is the automatic allocation of local variables into registers. This optimization is performed in three steps:

1. Find the set of local variables which never have their address taken.
2. Analyze the uses of these variables to compute their "lifetimes."
3. Allocate each available register to as many variables as possible, given the constraint that two variables with overlapping lifetimes cannot share the same register.

A variable whose address is taken cannot be placed in a register, since a pointer cannot point to a register. Avoid taking the address of commonly used variables where possible, even if this requires the use of a secondary variable.

Most variables in a C program have the characteristic that if they are referenced twice with no intervening stores, either direct stores or through pointers, then both references will deliver the same value. A variable is called "volatile" if it does not have this property. One way a variable can be volatile is if that variable is modified by an asynchronous interrupt handler. Another way that a variable can be volatile is because of memory mapped I/O.

By default, the InterTools optimizer assumes that only variables declared with the ANSI C `volatil`keyword are volatile.

WARNING: If you have not appropriately marked your volatile variables, then you **must** supply the `-vv` switch. This tells the compiler to assume that all global variables are volatile. This results in safe, but sub–optimal code.

Versions of the InterTools compiler earlier than 8.2 assumed that all globals were volatile by default and would drop the assumption under the `-nv` switch. For versions 8.2 and later, the old `-nv` behavior is the default and the `-vv` switch recreates the old pre–8.2 default.

The `-nal` option can be used to inform the compiler that named variables are not referenced indirectly through pointers. By default the optimizer must "forget" all its information about global variables at every store through a pointer, since it cannot know which variables are potentially pointed to or "aliased" with the pointer. In some programs all pointer variables point into the heap, so this loss of optimization is unnecessary.

Use caution when employing the `-nal` switch, since it is not safe for all programs. However, it can result in much improved code if used appropriately.

2.6 Optimizations Performed

Here is a brief summary of the transformations performed by the optimizer.

2.6.1 Automatic Register Variable Assignment

Most C compilers do not assign variables to registers unless they are declared with the C `register` keyword. The InterTools optimizer analyzes the usage of all local variables in a procedure and assigns them to registers in the optimal way. Furthermore, the optimizer may assign several variables to the same register, as long as the lifetimes of the variables are disjoint. Therefore the optimizer can do a better job than the most conscientious programmer.

Assigning variables to registers is the single most important key to generating good code for machines which have a large register set. This saves a load instruction at every reference, and a store at every assignment. Register variable allocation can reduce overall code size in programs which use a lot of local data.

Register allocation is done by a packing algorithm which takes into account the number of uses of each variable and whether these uses lie in a loop.

2.6.2 Common Subexpression Elimination

Two computations are called common subexpressions, or CSEs, if they are guaranteed to deliver the same value on all possible paths of program execution. The simplest example is two occurrences of the same local variable with no intervening store (either directly or through a pointer).

It is often better to compute a CSE once, save it in a temporary, and reuse the temporary at subsequent uses than to re–compute the CSE each time from scratch. This can save several instructions if the CSE is complex. Even if the CSE is very simple, like a 32–bit constant, it may be worthwhile to recognize the CSE just to decrease the size of the instructions which use that value.

The optimizer decides whether using a temporary is an improvement and transforms the program appropriately. The algorithm which makes this decision is quite sophisticated. It takes into account the availability of a register temporary, the number of times the CSE was used, and the cost to re–compute.

Common subexpression elimination is an extremely important optimization. It works especially well in conjunction with automatic register variable assignment, as the CSEs automatically become candidates for assignment to registers.

2.6.3 Target Path Computation

Often a computation is evaluated for the purpose of storing it into a particular register. The simplest example of this is an assignment into a register–resident variable. Here the register variable is called the "target" of the evaluation.

If the target register does not appear in the expression being evaluated, then the expression may be evaluated directly into the target register. For example, the "targeted" code for `a = b + c;` would be to move *b* into *a* and then add *c* into *a*. Of course, this code would be incorrect if *c* were in the same register as *a*. In contrast, the untargeted code would be to move *b* into a temporary register, add *c* to the register, and move the register into *a*. This requires an extra move instruction.

When the target register appears exactly once in the expression it may still be possible to find a target path which "rolls up" the expression in the target register. One example of this is the expression:

```
a = b + (a * c);
```

Here the targeted code would be to multiply *c* into *a*, and then add *b* into *a*. In contrast, the untargeted code would be to move *a* into a temp, multiply *c* into the temp, add *b* into the temp, and move the temp into *a*. This is twice as many instructions as the targeted sequence.

Targeting is an extremely valuable optimization because it occurs so often.

2.6.4 Strength Reduction

Strength reduction is an optimization that typically turns complex address calculations into additions inside a loop. In the classic case, the loop:

```
int a[10];
for (i=0; i<10; i++) {
   a[i] = 0;
}
```

can be turned into the equivalent of

```
long a[10],*pnt;
for (pnt = &a[0]; pnt<&a[10];) {
   *pnt++ = 0;
}
```

The generated code for the second loop is smaller and much faster.

2.6.5 Code Hoisting

A computation within a loop is called "loop invariant" if it is guaranteed to deliver the same value on each iteration through the loop. It is always faster to compute a loop invariant once outside the loop than to compute it on each iteration. This optimization is called code hoisting because the computation is effectively lifted out of the body of the loop.

In the case of nested loops, the hoisted code is examined again to see if it can be hoisted out of the enclosing loop as well. Of course, a computation can only be hoisted if it is guaranteed to be computed on all paths through the loop.

This optimization does not decrease the size of the generated code, but can greatly increase its speed. It is possible for code hoisting to slightly increase the size of the generated code, and so code hoisting can be disabled with a command line switch `-nh`.

2.6.6 Loop Rotation

A typical **for** loop is:

```
for (i = 1; i < 10; i++)
```

Much less common are loops of the form:

```
for (i = 1; i < j; i++)
```

The significant difference between these two loops is that the body of the first loop is guaranteed to execute at least once. Recognizing the difference between these two cases allows the compiler to save a jump instruction because it need not begin with the "test" code at the top of the loop. In other words, the loop may be "rotated" to perform the test at the bottom.

The above optimizations are performed in the optimizer phase but additional optimizations are performed in the back end phase itself. These optimizations are performed even if the optimizer phase is not run.

2.6.7 Branch Tables

Branch tables provide a more efficient implementation for switch statements. There are two main strategies of code generation for switch statements. One is for the generated code to compare the selector against each of the case labels in succession, and jump to the appropriate case when a match is found. This is called the "fall through" strategy. The other is for the generated code to use the selector as an index into a table of destination addresses, and then jump to the resulting address. This is called the "branch table" strategy.

The compiler automatically selects the best strategy for the particular switch statement at hand. The decision depends upon whether the switch table is "dense," that is, if most of the values between the lowest and highest case labels actually correspond to case labels. For example, suppose you had a switch statement with two case labels, one at zero and one at 10,000. It would be unwise for the compiler to implement this using a 10,000 entry branch table. In general, the compiler will make a branch table if the switch is at least one–third dense and has at least five cases. If the switch is less dense but has many cases, the compiler will choose its third option, a binary search run–time routine.

2.6.8 Entry/Exit Optimization

Most compilers typically emit a standard entry–exit sequence at the beginning and end of each procedure. This code consists of two parts: code to establish a stack frame, and code to save registers which must be preserved according to the run–time model. Both of these kinds of code are amenable to optimization.

First, the code to establish a stack frame is only necessary if the procedure being compiled uses stack data. Since the optimizer performs automatic register variable assignment, many procedures will

have only register–resident data, and thus do not need a stack frame. It should be noted that stack frames are necessary for the debugger to trace the stack, and so this optimization can be suppressed using the `-nl` command line switch.

Second, the code to save registers can be optimized to only save those registers which are actually used in the body of the subroutine. Furthermore, if only one register need be saved, then a single push instruction is used instead of a multiple push instruction.

In very small procedures, the entry–exit code can be a significant percentage of the entire time spent in the subroutine, so this can be a very important optimization.

2.6.9 Multiplication Optimization

Many compilers can recognize that a multiply by a power of two can be done by a shift. The InterTools compiler back end also recognizes many other multiplications by constants which can be done without using a multiply instruction. This is somewhat larger than a multiply instruction, but much faster.

One limitation of the MC68000 instruction set is the lack of 32–bit multiply and divide instructions. This means that the compiler must invoke a run–time library routine to perform long multiply and divide operations, at a considerable cost in execution time. However, if the compiler can determine that both multiplicands fit in a 16–bit signed word, then the compiler will use the `MULS` instruction and avoid the out–of–line call.

2.6.10 Subscript Optimization

The compiler uses 16–bit arithmetic for array index computations if it knows the dimension of the array and can determine that the computation will not overflow. This optimization is bypassed if the array has only one element, because it is obvious in such cases that the user is intentionally indexing off the end of the array.

2.6.11 Special Instruction Selection

The M68000 family has a number of special instructions which can be used to generate just the right code in particular circumstances. The InterTools compiler takes full advantage of these instructions. For example, the `DBcc` (decrement and branch on condition) instruction is appropriate for certain loops. The `CMPM` instruction is appropriate for the typical C expression `*p++ == *q++`.

2.6.12 Special Addressing Modes

The MC68020 and later targets have a number of powerful addressing modes which are not available on the MC68000. For example, array subscripts can be used without having to multiply them by the element size, as long as the element size is 2, 4, or 8. Also a pointer value may be used directly from memory without having to load it into a register.

The InterTools compiler takes full advantage of these addressing modes. Every memory reference is carefully analyzed so that the most efficient addressing mode may be chosen.

2.7 Messages

All C compiler error messages have the following format:

```
:XX:name.c:nnn:message
```

where:

- `XX` identifies the compiler phase which detected the error.

XX	Phase
`FE`	Front End
`OP`	Optimizer

- `name.c` is the name of the source file in which the error occurred.
- `nnn` is the line number in file *name.c* at which the error occurred.
- `message` is a description of the error.

Messages from the front end are graded in severity and are flagged either as (**Warning only**) or as (**FATAL!**). Severity levels for warning messages are listed below. Messages from the back end are **always** fatal. Messages from the optimizer may be warnings or fatal errors.

Note: C source files containing errors can cause the optimizer and back end to emit fatal error messages. Other errors from the back end are generally internal errors, and should be reported to Customer Support. Error messages and warning messages affect the system return code returned by the compiler.

Phase	Severity	Warning Message
FE	1	Address initialization not position–independent
FE	1	Block scope extern declaration may not have initializer
FE	1	Comma operator (",") not allowed in initialization
FE	1	Compilation defines no external names
FE	1	Degenerate unsigned compare with zero
FE	1	Duplicate qualifier
FE	1	Empty braced initialization list is illegal
FE	1	Empty character constant
FE	1	Extra comma at end of enumeration list ignored
FE	1	Filename is too long for list header
FE	1	Illegal hex constant (zero assumed)
FE	1	Missing ">" after header name (added)
FE	1	Non–int bitfield same as int bitfield
FE	1	Old style initializer
FE	1	Source line is too long for listing (truncated)
FE	1	Struct member redeclared
FE	1	Undefined, assuming 0
FE	1	Undefined, int function assumed
FE	1	Unsigned compare with negative constant
FE	1	Value specified is outside of legal range of 1 to 32767
FE	2	Case label not reachable by this type of switch selector

FE	2	Enum function returning member of different enum type
FE	2	Enum function returning value of different enum type
FE	2	Enum member assigned to variable of different enum type
FE	2	Enum member passed to prototype parameter of different enum type
FE	2	Enum variable assigned to variable of different enum type
FE	2	Enum variable passed to prototype parameter of different enum type
FE	2	Implicit conversion to enum type
FE	2	Name already defined – stmt ignored
FE	2	Negative or 0 array length
FE	2	Separate directive must come before definition
FE	2	Separate variable not declared
FE	2	Unreachable code
FE	2	Variable is already separate; directive ignored
FE	2	Variable never referenced
FE	3	"&" requires lvalue operand; "&" ignored
FE	3	#option sep_off without #option sep_on
FE	3	#pragma sep_off without #pragma sep_on
FE	3	Argument names ignored
FE	3	Can't specify two class names to a single separate segment
FE	3	Char constant too long

FE	3	Constant too big for field, value truncated
FE	3	Constant too big: truncated to 0
FE	3	Constant truncated to 1 byte
FE	3	Constant truncated to 2 bytes
FE	3	Constant value truncated
FE	3	Fixed point literal out of range
FE	3	Floating point constant truncated in conversion to integer
FE	3	Hex escape too long
FE	3	Hex escape truncated to four hex digits
FE	3	Hex escape truncated to two hex digits
FE	3	Integer constant truncated in conversion to floating point
FE	3	Invalid directory name
FE	3	Missing #include file name
FE	3	Negation of unsigned fixed point value
FE	3	Null or invalid directory name
FE	3	Syntax error in #option statement
FE	3	Syntax error in #pragma statement
FE	3	Too many –I directories
FE	3	Too many –S directories
FE	3	Unknown #pragma
FE	4	Comparison between void and non–void pointers
FE	4	Float/double switch expression truncated
FE	4	Implicit conversion of pointer

FE	4	Implicit conversion to pointer
FE	4	Incompatible pointer types
FE	4	Non–portable pointer comparison
FE	4	Structure operation on non–structure item
FE	4	Structure ref base not a pointer
FE	6	Constant variable never initialized
FE	6	Duplicate specification in _ASM predicate
FE	6	Empty formal parameter name list after _ASM keyword
FE	6	Empty formal parameter name list after _CASM keyword
FE	6	Malformed "always" _ASM predicate; identifier ignored
FE	8	Function declaration does not match previous function prototype
FE	8	Function prototypes present but not enabled in this compilation
FE	8	Name redefined
FE	8	Nonzero int assigned to pointer
FE	8	Nonzero int assigned to pointer return value
FE	8	Nonzero int used as pointer argument
FE	8	Pointer type in function call does not match function definition
FE	8	Pointer types in assignment do not match
FE	8	Pointer value in return does not match function type
FE	8	Prototype not compatible with previous non–prototype declaration

FE	8	Void parameter list disagrees with previous function prototype
FE	9	Division by 0 illegal; 1 assumed
FE	9	Division by 0.0 illegal; 1.0 assumed
FE	9	Negative shift count
FE	9	Shift count too large

CHAPTER 3

68000 Family Assembler

This chapter describes the usage of the InterTools assembler for the 68000 family of microprocessors. For more information about the assembly language, refer to the *Reference Manual* volume of the documentation set.

3.1 Introduction

Function: Assemble program(s) for the following targets:

Target	Invocation
MC68000	asm68000 *prog.asm* [*prog2.asm* ...] [*options*]
MC68EC000	asm68000 *prog.asm* [*prog2.asm* ...] [*options*]
MC68008	asm68000 *prog.asm* [*prog2.asm* ...] [*options*]
MC68010	asm68010 *prog.asm* [*prog2.asm* ...] [*options*]
MC68020	asm68020 *prog.asm* [*prog2.asm* ...] [*options*]
MC68EC020	asmec020 *prog.asm* [*prog2.asm* ...] [*options*]
MC68030	asm68030 *prog.asm* [*prog2.asm* ...] [*options*]
MC68EC030	asmec030 *prog.asm* [*prog2.asm* ...] [*options*]
MC68040	asm68040 *prog.asm* [*prog2.asm* ...] [*options*]
MC68EC040	asmec040 *prog.asm* [*prog2.asm* ...] [*options*]
MC68302	asm68302 *prog.asm* [*prog2.asm* ...] [*options*]
MC68330	asm68332 *prog.asm* [*prog2.asm* ...] [*options*]
MC68331	asm68332 *prog.asm* [*prog2.asm* ...] [*options*]
MC68332	asm68332 *prog.asm* [*prog2.asm* ...] [*options*]
MC68340	asm68340 *prog.asm* [*prog2.asm* ...] [*options*]
MC68360	asm68360 *prog.asm* [*prog2.asm* ...] [*options*]
MC68060	asm68060 *prog.asm* [*prog2.asm* ...] [*options*]

Input: *prog*`.asm` [*prog2*`.asm...`]

Output: *prog*`.ol [.lis, .xrf, .err, .gsm]`
[*prog2*`.ol...`]

3.2 Assembler Options: Summary

The assembler recognizes the following options:

Option	Function	See:
`-0`	(Zero) Display the version number of the executables.	3–17
`-a`	Generate source listing and show `INCLUDE`'d source.	3–7
`-A`	Force absolute addressing.	3–15
`-a4`	Force fullword alignment.	3–17
`-b`	Generate symbol table listing.	3–7
`-B`	Suppress listing of macro definitions.	3–7
`-bl`	Use 32–bit addressing for forward (undefined) branches. Not allowed for MC68000, MC68010 and MC68302 targets.	3–15
`-bs`	Use 8–bit addressing for forward (undefined) branches.	3–15
`-bw`	Use 16–bit addressing for forward (undefined) branches (default).	3–15
`-C`	Suppress listing of macro invocations.	3–7
`-d`	Generate symbolic debugging information.	3–18
`-e [`*erfn*`]`	Write all error messages to *erfn*; if *erfn* is omitted, suppress all messages.	3–8
`-err [`*file*`]`	**PC only.** Write error messages to *file*.	3–8
`-err+ [`*file*`]`	**PC only.** Append error messages to *file*.	3–8

Option	Function	See:
`-ex`	Allow the following TASKING extensions to the assembler language: `COMMON`, `ELSEC`, `ENDR`, `REPEATC`, `RESERVE`, `RESUME`, `RORG`, `%` (remainder) operator.	3–18
`-F`	Fold identifiers to upper case.	3–18
`-fs`	Use short addressing for forward (undefined) references.	3–16
`-g`	Generate global symbol listing.	3–8
`-h`	Allow use of hardware floating point instructions even with processors that do not support hardware floating point (MC68000, MC68010, MC68302, MC68332, MC68340 and MC68EC040).	3–19
`-I` *dir1* [*dir2...*]	Define `INCLUDE` directories.	3–14
`-l` [*lfn*]	Generate listing and send to file *lfn*; if *lfn* is omitted, write to *prog*`.lis`.	3–8
`-m`	Show all macro expansions.	3–8
`-M` *mfn*	Pre–`INCLUDE` file *mfn* into source stream.	3–15
`-N`	Suppress listing of conditional assembly directives.	3–8
`-o` *ofn*	Write object module to file *ofn*.	3–9
`-O`	Make a non–68000 assembler act like the 68000 assembler.	3–16
`-p`	Show code generated for structured syntax.	3–9

Option	Function	See:
-P [*lines*]	Set lines–per–page to *lines*; if *lines* is omitted, suppress pagination.	3–9
-po	Force PC–relative addressing in absolute section.	3–17
-ps	Force PC–relative addressing in relocatable section.	3–17
-s	Generate source listing and do not show INCLUDE'd source.	3–9
-S *dir1* [*dir2...*]	Define INCLUDE directories (equivalent to -I switch).	3–15
-t	Trim comments from listing.	3–9
-U	Show unassembled source.	3–9
-v	Verbose mode. Reports date, time, and status/result of assemble.	3–19
-V	Very verbose mode. Identifies executables as they are invoked. This determines which program was executing if the assembler aborts. For technical support use.	3–19
-w [*n*]	Suppress warnings of severity less than or equal to *n* (default 10).	3–19
-x	Generate cross–reference listing.	3–10

3.3 Usage

The assembler translates assembly language source programs into object modules. These object modules may be input to the linking locator or catalogued in a library. Source programs can also include (via `INCLUDE` options) other source files. Various listing options are available to display the results of assembly.

If an input file name with no extension is specified, e.g., *prog*, then the assembler will search for *prog* `.asm`.

The assembler recognizes the assembly language originally specified by Motorola, which is fully described in the *InterTools Reference Manual.*

The `-ex` option allows the use of the TASKING extensions. Please refer to the *Assembler Directives* chapter of the *InterTools Reference Manual* for details.

The assembler will produce object code for the MC68000, MC68010, MC68020, MC68030, MC68040, MC68302, MC68332, MC68340, MC68360, MC68EC030, MC68060 or MC68EC040 instruction set, depending upon the manner in which it is invoked. The assembler will disallow instructions which do not exist on the specified target processor. The resulting object module is labeled internally with the target name, e.g., object modules produced by `asm68020` are labeled as containing "68020" code. The linking locator will issue a warning if you attempt to link object modules intended for different targets. Note that the use of MC68851 memory management coprocessor instructions is allowed only with `asm68020`.

Note: From now on, we will refer to the assembler as the 68000 assembler.

Example

Assemble with all defaults:

```
asm68000 test.asm
```

- Assemble `test.asm`.
- Search for any `INCLUDE` files in current directory.
- Write object module to file `test.ol`.
- No listings will be generated.

3.4 Assembler Options: Detailed Descriptions

This section describes the assembler options in more detail and provides examples of their use.

3.4.1 Listing Options

The listing options control the generation of the various listing files. Listings are not produced by default; you must specify the appropriate option or options.

`-a`	Generate a source listing showing secondary `INCLUDE`'d lines in addition to primary source lines.
`-b`	Generate symbol table and macro definition listing.
`-B`	Suppress listing of macro definitions.
`-C`	Suppress listing of macro invocations.

`-e` [*erfn*]
: If *erfn* is specified, then error messages are directed to *erfn*. Error messages are also written to the listing file, if any. If *erfn* is omitted, then all error messages are suppressed. The default is to print error messages to stderr.

`-err` [*file*]
: **PC only.** Write error messages to file *file*. If *file* does not exist, it will be created. If *file* does exist, it will be overwritten. If *file* is omitted, then error output will be redirected to standard output.

`-err+` [*file*]
: **PC only.** Just like `-err`, except output will be appended if *file* exists.

`-g`
: Generate global symbol listing, using the `gsmap` utility. If `-l` is not specified, the listing is written to file *prog*`.gsm`.

`-l` [*lfn*]
: This switch controls the destination of the listing output implied by the other listing switches. If *lfn* is given, then the listing is written to file *lfn*. *If lfn* is +, then the listing is written to standard output. If *lfn* is omitted, then the listing is written to file *prog.lis*. If multiple source files are given in one assembler invocation, then neither + nor *lfn* may be specified. Instead, a separate listing file is generated for each input file. The listing output corresponding to *progx.asm* appears in *progx*`.lis`.

`-m`
: Show the expansion of all macros in the source listing.

`-N`
: Suppress listing of conditional assembly directives.

`-o` *ofn*	If *ofn* is specified, then write the object module to file *ofn*. If *ofn* is +, then the object module is written to standard output. The default is to write to file *prog*`.ol`.
`-P` [*lines*]	Set the number of lines per listing page to *lines*. If *lines* is omitted, then suppress listing pagination.
`-p`	Show code generated for structured syntax, e.g., `IF-THEN-ELSE`.
`-s`	Generate a source listing. Secondary `INCLUDE`'d lines are not listed.
`-t`	Trim comments from listing.
`-U`	List unassembled source, that is, lines that are excluded via conditional assembly constructs.

`-x`

Generate cross–reference listing. If `-l` is not specified, the listing is written to file *prog*`.xrf`.

If no listing option is specified, then no listing is generated. If only one of `-a`, `-b`, `-s`, or `-x` is specified, then only that particular listing is generated. Use a combination of options to obtain a listing containing more than one of the listing types described above.

If the `-l` switch is used (with or without an explicit listing filename) and **none** of `-a`, `-b`, `-s`, or `-x` is also supplied, then a listing showing only primary source is generated. The following invocations of the assembler are thus all equivalent:

```
asm68000 prog.asm -s
asm68000 prog.asm -l
asm68000 prog.asm -s -l
```

Example

Generate source and cross–reference listings:

```
asm68000 spiral2.asm -x -s
```

- Assemble `spiral2.asm`.
- Write object module to file`spiral2.ol`.
- Write source listing to file `spiral2.lis`.
- Write cross reference listing to file `spiral2.xrf`.

Partial source of `spiral2.asm`:

```
XTRAP     MACRO
          IFC       '\1','CHAR'
          LEA       CHAR,A5
          LEA       CHAREND,A6
          ENDC
          TRAP      #fifteen
          DC        seven
          ENDM
*****************************************
* equ's
*****************************************
one       EQU       1
fifteen   EQU       15
seven     EQU       7
          PAGE
          SECTION   TEXT
S  TART:
          CLR       D0
          LEA       CLEAR,A5
          LEA       CLREND,A6
          XTRAP     NOCHAR
* ERASE:
          LEA       CENTER,A5
          LEA       CENEND,A6
          XTRAP     NOCHAR
*
          MOVE      #one,D2
```

Partial source listing in `spiral2.lis`:

```
Source   file: spiral2.asm

16 0                        | XTRAP      MACRO
17 0                        |            IFC
'\1','CHAR'
18 0                        |            LEA  CHAR,A5
19 0                        |            LEA
CHAREND,A6
20 0                        |            ENDC
21 0                        |            TRAP #fifteen
22 0                        |            DC   seven
23 0                        |            ENDM
24 0                        |
************************
25 0                        |  *  equ's
26 0                        |
************************
27 0      [$1]              |  one       EQU      1
28 0      [$F]              |  fifteen   EQU      15
29 0      [$7]              |  seven     EQU      7
31 0      >                 |            SECTION TEXT
32 0                        |  START:
33 0      4240              |            CLR   D0
34 2      4BF9{00000000}    |     LEA    CLEAR,A5
35 8      4DF9{00000002}    |     LEA    CLREND,A6
36 E                        |            XTRAP NOCHAR
37 12                       |  *
38 12                       |  ERASE:
39 12     4BF9{00000002}    |          LEA   CENTER,A5
40 18     4DF9{0000000A}    |          LEA   CENEND,A6
41 1E                       |             XTRAP NOCHAR
42 22                       |  *
43 22     343C0001          |             MOVE  #one,D2
```

Partial cross–reference listing in `spiral2.xrf`:

```
Feb 12 1997 10:13:22 CROSS-REFERENCE: spiral2.asm PAGE 1
   BAKEND
      Def : spiral2.asm 103
      Ref : spiral2.asm 65

   CENEND
      Def :  spiral2.asm   99
      Ref :  spiral2.asm   40

   CENTER
      Def :  spiral2.asm   97
      Ref :  spiral2.asm   39

   CHAREND
      Def : spiral2.asm 119
      Ref : spiral2.asm 51 59 67 75

   CLEAR
      Def :  spiral2.asm   93
      Ref :  spiral2.asm   34

   CLREND
      Def :  spiral2.asm   95
      Ref :  spiral2.asm   35

   FOREND
      Def : spiral2.asm 107
      Ref : spiral2.asm 49

   FORESP
      Def : spiral2.asm 105
      Ref : spiral2.asm 48

   fifteen
      Def :  spiral2.asm   28
      Ref :  spiral2.asm   36  41  50  51  58  59
                   66 67 74 75
```

Example

Specify a different filename for the object module:

```
asm68000 myprog.asm -o myobj.ol
```

- Assemble `myprog.asm`.
- Write object module to file `myobj.ol`.

3.4.2 INCLUDE Options

`-I` *dir1* [*dir2* ...]

Define directory(ies) to be searched for user include files. The default is to search the current working directory. No more than 32 user include directories may be specified.

`INCLUDE` *file*

`INCLUDE` <*file*> The first form is considered to be a user include; the second form is considered to be a system include.

When searching for user includes, the assembler first searches the directories specified by the `-I` switch(es), followed by the directories specified by the `-S` switch(es), followed by the current working directory.

When searching for system includes, the assembler only searches the directories in the order specified by the `-S` switch(es).

-M *mfn*	Pre–INCLUDE file *mfn* into source stream. The named file is INCLUDE'd before any of the source file is processed. Thus, if a library of macros is written, they can be predefined just as if the first line of the source file were INCLUDE *mfn*.
-S *dir1* [*dir2...*]	Define directives to be searched for system include files. There is no default. No more than 32 system include directories may be specified. The following are some INCLUDE directories as they might appear in an assembly program source file.

3.4.3 Code Generation Options

-A	Generate absolute addresses instead of PC–relative addresses whenever possible. Code generated with this option will take up more space because the address occupies 32 bits, while a PC–relative displacement occupies either 8 or 16 bits.
-bl	Use 32–bit addressing for forward branches. This switch cannot be used when assembling for the MC68000, MC68010 or MC68302 targets.
-bs	Use 8–bit addressing for forward branches. If the displacement does not fit in one byte, the assembler will issue an error message and try to generate code anyway.
-bw	Use 16–bit addressing for forward branches (default).

`-fs` Use short (16–bit) addressing for forward references other than branches. If this option is not specified, long (32–bit) addressing will be used.

`-O` Make a non–68000 assembler act like the 68000 assembler.

Sometimes the assembler must choose an instruction form when it doesn't have enough information to pick the optimal legal form. For example, consider the instruction "`MOVE #1,(xxx,A0)`", where "xxx" is an external name defined in an XREF directive. The assembler must decide how much space to allocate to hold the offset value (xxx). When assembling for the MC68020 target, there are two choices: 16 bits or 32 bits.

In these situations, the assembler generally picks a less optimal form that is more likely to execute properly than a more optimal form that might not be suitable. In that spirit the assembler tends to pick long–form instructions when it cannot be sure that the corresponding short–form instruction would be sufficient. Thus, in the example above, the 68020 assembler would choose a 32–bit offset.

The effect of the `-O` switch is to restrict the assembler in these situations to behave as if only the 68000 base instruction forms were available to choose from. In the above example, the 68000 assembler would choose a 16–bit offset, because the 68000 does not support the

32–bit offset addressing form.

This switch is appropriate for assembler source that is known to assemble properly through a 68000 assembler. In that case it is possible that an assembler for another target (like the 68020) might make less optimal code, since it's trying to be safe.

`-po` Use PC–relative addressing in absolute sections. This applies even to labels in sections other than the current one. Labels in the current section are normally referred to via PC–relative addressing, unless the displacement does not fit in 16 bits.

`-ps` Use PC–relative addressing in relocatable sections. This applies even to labels in sections other than the current one. Labels in the current section are normally referred to via PC–relative addressing, unless the displacement does not fit in 16 bits.

3.4.4 Miscellaneous Options

`-0` (Zero) Display the version number of executables. For technical support purposes.

`-a4` Force fullword alignment.

By default the assembler assigns each segment halfword alignment. This switch forces all generated segments to have fullword alignment. Since the MC68020, MC68030, MC68040, MC68360, MC68EC030, MC68EC040, and MC68060 have a 32–bit data bus, fullword memory accesses take less time on fullword aligned addresses.

`-d` Include symbolic information in the object module. The default is no symbolic debugging information. The linking locator and formatter programs pass symbolic information through to their output files. Eventually the symbol information will reside in a hex output file, or in a debugger symbol file to be read by CrossView Pro. Symbolic information can also be displayed with the symbol list program, symlist. (See the *Symbol List Utility* section of the *Other Utilities* chapter.)

Example

Include symbol table information:

```
asm68000 myprog.asm -d
```

- Assemble `myprog.asm`.
- Write output object module to file `myprog.o1`.
- Include symbol table information in the output object module.

`-ex` Allow the following TASKING extensions to the assembler language: `COMMON`, `ELSEC`, `ENDR`, `REPEATC`, `RESERVE`, `RESUME`, `RORG`, `%` (remainder) operator.

`-F` Fold identifiers to upper case.

`-h`	Allow use of MC68881 floating point coprocessor instructions. By default MC68881 instructions are accepted when assembling for the MC68020, MC68030, MC68040, MC68060 or MC68EC030 target. This switch can be used to force the assembler to accept MC68881 instructions when assembling for the MC68000, MC68010, MC68302, MC68332, MC68340, MC68360, and MC68EC040 targets.
`-v`	Verbose mode. Reports date, time, and status/result of assemble.
`-V`	Very verbose mode. Identifies executables as they are invoked. This determines which program was executing if the assembler aborts. For technical support use.
`-w` [*n*]	Suppress warning messages of severity less than *n*. Warning severities vary from 1 to 9, (1 = least severe to 9 = most severe) depending upon the error. If omitted, *n* defaults to 10, i.e., all warning messages are suppressed. The default is to issue all warning messages.

3.5 Messages

A list of all the assembler error messages appears below. Warning messages may be suppressed using the `-w` switch.

Warnings

ASCT cannot be specified for an external symbol.

Ambiguous use of index registers; assuming post_index mode.

Assembler error in branch–relative processing.

Attempt to divide by 0.

Attempt to redefine *name* as a string variable.

Attempt to redefine EQU symbol: *name*.

Attempt to redefine a label *name*.

Attempt to redefine a macro: *name*.

Attempt to shift right <number> bits.

Attempt to use EQU symbol *name* as a label.

Attempt to use REPEAT_ENDR construct outside of macro.

Attempt to use SET symbol *name* as a label.

Attempt to use a reserved word as a label (*name*).

Attempt to use a section name (*name*) as a symbol.

Attempt to use an ELSE in a REPEAT block.

Can't open temporary cross–ref file *name*.

Cannot EQU to register plus offset (*name*).

Cannot find include file *name*.

Consecutive ELSEs not allowed.

Constant value will not fit in a byte.

Constant value will not fit in a word.

Directive *name* is not supported.

Displacement too far for specified size.

Ending <character> assumed.

ENDM encountered outside of macro definition.

ENDOF function result.

EXITM encountered outside of macro invocation.

Expected close of *name* block.

Extraneous argument (*name*).

Illegal character in digit string: <character>.

Illegal character in input stream, octal *name*.

Illegal operand for CMP.

Illegal operator for relocatable expression.

Illegal operator for relocatable expression: <character>.

Illegal redefinition of section name *name*.

illegal relocatable operand value.

Illegal size for this mode.

illegal size specification.

Illegal structured syntax.

illegal trap vector value.

Improperly nested conditional assembly construct.

Include nesting overflow – limit is <number>.

Include pathname is missing.

Incomplete string.

Invalid conditional code for the *name*.

invalid index register size.

Invalid macro argument: *name*.

invalid macro argument specifier: <character>

invalid mnemonic.

Invalid register size specifier: <character>.

Invalid size for CAS2.

Invalid size for CHK.

Invalid size for data item.

Invalid size for relocatable data item.

Invalid size specifier: <character>.

Line too long: truncated to <number> characters.

Long size is not valid for this processor.

Macro *name* used before defined.

No code–generating directives may appear in an OFFSET segment.

No code–generating instructions may appear in an OFFSET segment.

No floating point software.

No register size allowed.

No register size specifications allowed.

no size specification allowed.

No size specification allowed on the first register.

Numeric SET symbol *name* referenced before defined.

only byte size allowed.

only long size allowed.

Only one index register is allowed; using first as base register.

only word size allowed.

Only word and long size allowed.

only word or long size allowed.

Option *name* is meaningless.

Option *name* is not supported.

Postincrement mode must use an address register; Assuming A <number>.

RESERVE sections cannot have absolute locations.

RESUME: *name* is not a section.

RORG used in absolute section: treated as ORG.

Redefining keyword *name* as a macro.

Register mask ranges must go from low to high.

SET symbol *name* not found.

Scalar value required here.

Standard include pathname is missing.

String in *name* exceeds declared length.

Subfield <number> in command switch field <number> is <=0.

Symbol *name* cannot be made global.

Symbol *name* is defined in this module.

Symbol *name* must be global for use in ENDOF.

Symbol *name* never defined.

Syntax error near token *name*.

Syntax error near token <EOL>.

The first operand must have a value between 0 and 7.

The first register must be an address register or PC.

The scale for an index register must be 1, 2, 4, or 8.

There is no *name* statement to match.

This addressing mode is not valid for this processor.

This directive is not supported.

This instruction, mode, or size is not valid for this processor.

This instruction is not valid for this processor

This instruction is not valid for the 68040

This instruction is not valid for the 68EC030

This instruction is not valid for the 68EC040

This instruction is valid only for the 68030 or 68851

Too many include pathnames – limit is <number>.

Too many standard include pathnames – limit is <number>.

Truncation occurred in string to int conversion.

Unclosed *name* statement.

Unclosed FOR statement. Assuming ENDF.

Unclosed REPEAT statement. Cannot recover.

Unclosed WHILE statement. Assuming ENDW.

Unknown operator (<character>).

Unmatched IF, WHILE, REPEAT, of FOR statement.

Unrecognized character: <character>.

Unrecognized condition code: *name*.

Unrecognized register.

Errors

Assembler bug in macro definition: macro name lost.

Assembler error in generation of REPEAT label.

Assembler in symbol lookup: *name*.

Can't allocate enough space.

Can't install *name* as a segment.

Can't open error file *name*.

Can't open listing file *name*.

Can't open object file *name*.

Can't reopen primary source file *name*.

Cannot open source file *name* or *name*.

Cannot open source file *name*.

End of file *name* encountered during macro definition.

Improperly nested REPEAT_ENDR block.

Macro nesting depth >500 – probable infinite recursion.

No more space in symbol table.

No source file specified.

No space left in macro table for *name*.

No space left in symbol table for *name*.

No space left in symbol table.

SECTION directive must reference a section.

CHAPTER 4

Linking Locator

This chapter describes the operation and use of the Linking Locator utility. It begins with a summary listing of the available options and continues with more detailed explanations of their usage, linking concepts, compiler runtime libraries, library searches, locator commands, and error messages.

4.1 Introduction

Function: Combine object modules, create ROM–able initialization segment, assign absolute addresses to segments.

Invocation:

`llink` [*prog*.[`ol` | `ln` | `rmp`]...] [*options*]

Input: Object modules and locator commands

Output: Standard output (or *prog* . `ab` or *prog* . `ln` or *prog* . `rmp`)

Note: The `llink` linking locator is for C modules only. See the *InterTools C++ User's Guide* for more information on `ldriver`, the C++ linking locator utility.

4.2 Linking Locator Options: Summary

The linking locator recognizes the following options:

Option	Function	See Page:
`-0`	(Zero) Display the version number of executable.	4–16
`-b` *segname*	Specify the segment to be created. Default output segment is `rompOutSeg`.	4–12
`-c` *cfn*	Read locator commands from file *cfn*.	4–10
`-err` [*file*]	**PC only.** Write error messages to *file*.	4–16
`-err+` [*file*]	**PC only.** Append error messages to *file*.	4–16
`-G`	Suppress all global symbols in output file.	4–14
`-i` [*ifn*]	Take the names of input object modules from file *ifn*. If *ifn* is omitted, read names from standard input.	4–16
`-il` *ifn*	Read library index file name(s) from file *ifn*.	4–9
`-k` [*sym sym2...*]	Keep only the named global symbols in output.	4–14
`-L` *lib* [*lib2...*]	Specify library index file(s) to be searched.	4–9
`-lo`	Suppress locate processing (link only).	4–10

Option	Function	See Page:
`-o` [*ofn*]	Write output to file *ofn*. If *ofn* is omitted, write to *prog*.`ab` if locate processing is performed, otherwise to *prog*.`rmp` if ROM processing is performed, otherwise to *prog*.`ln`. If the switch is omitted, write to standard output.	4–17
`-opfile` *opts*	Supply command line options in a file *opts*.	
`-p` *n*	Pad the size of all segments by *n* bytes.	4–10
`-p` *n%*	Pad the size of all segments to *n* percent of their original size (*n* must be > 100).	4–10
`-rc` *class1* [*class2...*]	Create initialization segment for all segments of the named class(es).	4–12
`-rs` *seg1* [*seg2...*]	Create initialization segment for the named segment(s).	4–12
`-s` [*sym sym2 ...*]	Suppress the named global symbols in output file.	4–14
`-S`	Suppress all local symbols in output file. Symbols are generated by the compiler or assembler when `-d` is used.	4–15
`-v`	Report linking actions as performed.	4–17

Option	Function	See Page:
-w	Suppress warning messages, e.g., for unresolved references.	4–17
-x	Create external references for Cross-View Pro runtime support routines.	4–18

4.3 Usage

The linking locator performs any combination of three basic functions. These functions are called linking, locating, and ROM processing.

By default, `llink` performs only the linking and locating steps. ROM processing is performed only if one of the ROM processing options, `-b`, `-rc` or `-rs` is specified. If the `-lo` switch is specified, the locate step is bypassed.

4.3.1 Linking

The link step consists of combining linked or unlinked object modules into a single output module. References between the input modules are resolved during linking.

The modules to be combined may be named on the command line or listed in a file presented to `llink` via the `-i` switch. Modules listed in the file should be listed one per line followed by a carriage return. Be aware that MS–DOS enforces a relatively low limit on the number of characters in a command, generally about 128 or less. If + is given as an input module name, standard input is read. `Llink` attempts to resolve any undefined symbols by searching the given library index files for modules which define the symbols. External references which cannot be found in the given libraries are reported as warnings.

It is possible to "pre–link" part of a system and resolve remaining external references in subsequent links. For example, if one module is being changed and tested, the remaining object modules can be linked without the module in question. Each revision of the test module can be linked with the pre–linked portion. This speeds up the linking process and simplifies the `llink` command line.

4.3.2 ROM Processing

A wide class of embedded applications need to begin (or restart) execution without loading (or reloading) memory from an external device such as a disk. Such applications are called "ROM–based" applications, since the program must reside permanently in ROM (read–only memory).

All ROM–based systems must execute code to initialize their read–write data, since the initial values cannot be maintained in RAM (random–access memory), and read–write data cannot be allocated in ROM. ROM processing is a feature which simplifies and automates the data initialization process.

One technique to initialize global data is to code an explicit assignment statement for each individual global variable, and *never* code an initial value specification on a global data declaration.

When using ROM processing, initial values may be coded in the source on declarations as needed. The compiler places the initial values corresponding to all initialized non–separate variables in the `idata` segment. The simplest form of ROM processing consists of reading an object module and producing another module which is identical, except that:

1. The idata segment contains no initial values.
2. A new segment named **`rompOutSeg`** has been added.

Unlike `idata`, the **`rompOutSeg`** segment is suitable for placement in ROM. It contains a recipe for initializing the `idata` segments as indicated in the input module. Basically, this recipe consists of a sequence of triples of the form "address–length–data." This kind of segment is called an "initialization segment."

Note: Segments are explained in more detail in the *Linking Concepts* section below.

The `rcopy` runtime library routine knows how to follow a recipe in this format. It expects to receive the address of an initialization segment as a parameter. When `rcopy` is called, it follows its recipe, which results in the values in **`rompOutSeg`** **(in ROM)** being copied into `idata`, (in RAM). The user's system start–up or reset code must call `rcopy` when appropriate. Typically this is done at the start of the C `main routine`. However, when building a C++ application, ROM processing should be performed before `main` is called, i.e. in the assembly language system initialization file that calls `main`. An example of this method is provided in the ROM procesing Options section of this chapter.

This is necessary because the C++ compiler creates instructions at the start of main to invoke the constructors of statically–allocated objects. The constructors execute before the instructions corresponding to the first source line of main. If ROM processing were not performed until after such constructors run, the constructors would read uninitialized ROM memory. Similarly, any writes to initialized memory would be lost when ROM processing finally occurred.

Any list of input segments can be processed, so separate variables and assembly language segments can also be initialized. The name of the output segment can be specified using the `-b` switch (**`rompOutSeg`** is the default).

ROM processing can be performed several times, but the user program must include as many calls to `rcopy` as there are initialization segments.

You can omit the ROM processing step if there is no initialized data in your system. Note, however, that the runtime library contains several potential sources of initialized data.

For a detailed example of ROM processing, refer to the *Introduction to System Building Concepts* section in the *Tutorial* chapter of the *Getting Started Manual*.

4.3.3 Locating

The locate step consists of assigning target–machine addresses to the code and data contained in the input module(s) and resolving address references between segments accordingly. This process is done by obeying optional user commands or default rules. These commands are described in the *Locator Commands* section.

The result of locating is called an "absolute" module, because no relocatable references remain. Absolute modules are suitable for input to the formatter.

In general it is not possible to link an absolute module with other object modules, because absolute segments cannot be combined. Refer to the *Segments* part of the *Linking Concepts* section for more details.

Example

Link, locate and ROM process:

```
llink myprog.ol rest.ol -c sys.lc -rs idata -o
```

- Combine object modules `myprog.ol` and `rest.ol`.
- Locate according to commands in `sys.lc`.
- No library index files are searched for unresolved externals.
- Generate the segment **`rompOutSeg`** for initialization of the `idata` segment.
- Write absolute linked module to `myprog.ab`.

After the description of the `llink` options, this chapter contains a discussion of basic linking concepts, an overview of the compiler runtime library, a description of the library search algorithm, and a description of the available locator commands.

4.4 Linking Locator Options: Detailed Descriptions

This section describes the linking locator options in greater detail and includes examples of their use.

4.4.1 Linker Options

`-il` *ifn*

Read library index to be searched from file *ifn*. Index file *ifn* lists all libraries that would be specified on the command line if the `-L` switch were used.

On the PC, if the library name is a simple file name and it is not found in the current directory, llink will search the directories specified in the environment variable "`LIB`." The format of the `LIB` environment string is the same as the MS–DOS path variable. This variable may also be named `I2LIB` to avoid conflicts with the other software.

`-L` *lib* [*lib2...*]

Name library index files to be searched for unresolved externals. If the index file indicates that a given external can be resolved by reading a particular module, that module is included in the link. The *Librarian* chapter explains how library files are built and managed. If a module name in the library index file is not a full pathname, llink searches for the module in the directory containing the index file.

`-opfile` *opts*

This switch causes the linker to read command line options from file *opts*.

Compiled code must be linked with the runtime library supplied with the InterTools distribution. See the *Compiler Library Organization* section for information about the compiler runtime library.

Note: The linker portion of the linking locator may not always search the libraries in the order given. See the *Library Searches* subsection for more details.

4.4.2 Locator Options

Locate processing is done by default. If the `-lo` switch is present, locate processing is not performed.

`-c` *cfn*
: Read locator commands from file *cfn*. See the *Command Descriptions* section in this chapter for more information about locator commands.

`-lo`
: Suppress locate processing (link only).

`-p` *n*
: Pad the size of all segments by *n* bytes. This is equivalent to the following locator command:

```
SEGSIZE ( n ) ;
```

`-p` *n%*
: Pad the size of all segments to *n* percent of their original size (*n* must be > 100). This is equivalent to the following locator command:

```
SEGSIZE ( n % ) ;
```

Note: On the PC, if you use this in a `.BAT` file, remember that you must use two % signs because of MS–DOS syntax rules.

Example

Link only:

```
llink myprog.ol test.ol nph.ol -lo -o
```

- Link object modules `myprog.ol`, `test.ol`, and `nph.ol`.
- Write linked module to `myprog.ln`.
- Write warnings for unresolved references.
- No library index files are searched for unresolved externals.
- No ROM processing is performed.
- No locate processing is performed.

Example

Locate only:

```
llink myprog.ln -c sys.lc -o
```

- Read object module `myprog.ln`.
- Read locator commands from `sys.lc`.
- Write absolute linked module to `myprog.ab`.

4.4.3 ROM Processing Options

ROM processing is performed if and only if some ROM processing option is present.

`-b` *segname*	Specify the name of the segment to be created. The default name is **`rompOutSeg`**.
`-rc` *class1* [*class2...*]	Specifies that all segments of the named class(es) will be processed.
`-rs` *seg1* [*seg2...*]	Specifies that the named segment(s) will be processed.

Example

ROM processing only:

```
llink myprog.ln -rs idata -rc isep -lo -o
```

- Read object module `myprog.ln`.
- Process the segment named `idata` and any segments of class `isep`.
- No locate processing is performed.
- Write modified object module to `myprog.rmp`.

Example

Assume we have performed (or plan to perform) ROM processing with the -b switch supplied with the segment name _my_rompseg. Here is a sample C program that invokes rcopy:

```
#pragma separate my_rompseg
extern int my_rompseg;
#include <rcopy.h>
main ()
{
   rcopy (&my_rompseg);
...
}
```

Coding the external variable declaration is a technique used to induce the compiler to pass a pointer to the initialization segment to the rcopy routine. There is no actual variable named my_rompseg.

Note: This technique cannot be used without the -b switch. The compiler prepends an underscore to the C source name, my_rompseg, to form the linker global symbol name, _my_rompseg, referenced at the call statement. See the *Linking C and Assembly* application note and the *Compiler Naming Conventions* appendix for more details.

Note: Also note that the call to rcopy is the first thing executed by the user program.

Example

Assume that this application uses C++. We have performed ROM processing with the `-b` switch supplied with the segment name `_my_rompseg`. In this case we want to invoke rcopy from our system initialization file just prior to the jump to main. Here is a sample C program that performs this:

```
XREF _rcopy
      .
      .
      .
PEA _my_rompseg    ; Push address of _my_rompseg
JSR _rcopy         ; Call _rcopy
ADDQ #4, A7
      .
      .
      .
BSR.L _main
```

4.4.4 Symbol Options

`-G`	Suppress all global symbols in output file. This is equivalent to `-k`; it is only retained for backwards compatibility.
`-k` [*sym sym2...*]	Keep only the named global symbols in the output module; suppress all others. If no symbols are named, suppress all global symbols (this is equivalent to the `-G` switch).
`-s` [*sym sym2...*]	Suppress the named global symbols in the output module; keep all others.

`-S` Suppress all local symbols in output file. Debugging symbols are generated by the compiler or assembler when `-d` is used.

These switches control the retention of global and local symbols in the output file. The default is to retain all symbol information.

The "local" information refers to that which is added by the `-d` compiler and assembler switches. These symbols play no part in the linking process; they are only present for debugging purposes.

Global symbols are generated by the compiler and assembler for global variables and procedures. The compiler's rules for forming global symbol names are described in the *Compiler Naming Conventions* appendix. Note that the names specified in `-s` and `-k` must be those formed via these conventions.

Generally all global symbols must be retained in the output module to permit any further references to be resolved during later links. Specific global symbols may be suppressed to mask name conflicts. The switches which apply to global symbols are mutually exclusive.

If no debugging is intended and the link is complete, all symbols may be stripped. Stripping symbols reduces the amount of disk space required to hold the output module and speeds up the execution of llink and the formatter. It does not affect the size of the user program or the download hex file generated by the formatter.

Note: Suppressing either global or local symbols will prevent the formatter from creating symbolic records for CrossView Pro debugger symbol files.

Example

Suppress specified global symbol in output file:

```
llink compute.ol rah.ln -s _double -lo -o
```

- Link `compute.ol` and `rah.ln`.
- Suppress global symbol `_double` in `compute.ln`.
- No locate processing is performed.
- Write linked module to `compute.ln`.

4.4.5 Miscellaneous Options

`-0` (Zero) Displays the version number of the executable (for technical support purposes).

`-err` [*file*] **PC only.** Write error messages to file *file*. If *file* does not exist, it will be created. If *file* does exist, it will be overwritten. If *file* is omitted, error output will be redirected to standard output.

`-err+` [*file*] **PC only.** Just like `-err`, except output will be appended if *file* exists.

`-i` [*ifn*] This switch specifies that the names of input object modules are to be taken from the file *ifn*. The input module names should be listed in the file, one per line. Comments may be placed in the file by starting a comment line with "--". The name of the first module listed will be used as a default for constructing the name of the linked output file. If *ifn* is omitted, the names of the files are read from *stdin*.

`-o` [*ofn*]

This switch specifies the name of the output file. If *ofn* is omitted, write to *prog*.`ab` if locate processing is performed, or to *prog*.`rmp`, if ROM processing is performed, or to *prog*.`ln`. If the switch is omitted, write to standard output. Here the *prog* base name comes from the first input object module, whether named on the command line or in a file supplied via `-i`.

`-v`

Verbose mode. Reports the following linking actions as performed:

— The names of the object modules being read

— The names of the library index files being searched.

— The name of the output module.

`-w`

This switch inhibits warning messages. If `llink` is not performing the locate function, the "unresolved externals" warning is the only warning message that `llink` can emit. This can safely be suppressed if unresolved external references are expected. Other warning messages represent error conditions and should not in general be ignored or suppressed.

`-x` Forces the creation of external reference for the symbols `BREAKPT` and `__end__`. This causes the runtime library defining this symbol to be brought into the link. This is necessary when the program being linked will be run under CrossView Pro. See the *CrossView Pro User's Manual* for more information.

Note: Do not use the llink `-x` switch with ROM Monitor versions of CrossView Pro. The `-x` switch, which is used to build programs to be debugged with emulator–based versions of the debugger, will link in the runtime library object modules `end.ln` and `breakpt.ln`. Although `end.ln` should be linked with the application, `breakpt.ln` will interfere with the way the ROM Monitor handles code breakpoints. Instead, `end.ln` should be linked in explicitly on the llink command line. (`end.ln` contains code which allows you to take advantage of CrossView Pro's ability to evaluate function calls on the debugger command line.)

4.5 Linking Concepts

The following section defines some technical terms which are used in the descriptions of the linking locator functions and commands.

4.5.1 Segments

Target–memory in a linked relocatable module is represented as a set of "segments." A **segment** is an indivisible unit representing a sequence of contiguous target memory words which can be located at some absolute target address. Segments are defined directly by the user in assembly language or implicitly by the compiler when processing C programs. Each segment has a number of attributes; most important are its name, its length (in bytes of target memory) and its binary initial values. Other attributes include its memory space, combinability, and **class** membership. These are discussed below in greater detail.

The initial values of a segment consist of code (machine instructions) or data. The initial values are not required to define all the bytes contained in a segment. This is the case, for example, with uninitialized storage defined in assembly language. When a segment is loaded into memory, any uninitialized bytes retain whatever value they had before the program was loaded.

Normally the linking locator combines individually declared data items from different compilation units into common data segments. However, the InterTools compiler supports an extension to the C language, the compiler directive `#pragma separate`. This feature forces specified global data items to be placed into specified segments. These segments can then be assigned specific absolute target memory addresses by using the `locate` command. The `#pragma separate` feature is described in the *Pragma Separate (Option Separate)* application note.

For a full description of the segments created by the compiler, see the *Compiler Naming Conventions* appendix.

Example

This example will show you how to locate a memory–mapped I/O variable.

Suppose there is an 8–bit memory mapped I/O port at address 100 (decimal). In a C source program, define a character variable for the I/O port as follows:

```
#pragma separate io_port
char io_port;
```

The compiler allocates the variable `io_port` in its own segment named `S_io_port` The locator command to position `S_io_port` at address 100 is:

```
locate ( S_io_port : 100 );
```

MC68000 family processors access data items larger than a byte more efficiently if they are located at an even address. The MC68000 cannot load data items larger than a byte from odd addresses without causing an addressing exception. Other processors can do so, but less efficiently. The "alignment" attribute of a segment passes this information from the compiler or assembler to llink. For example, the compiler specifies word alignment for segments containing word–type variables, and llink will refuse to locate a segment of word alignment at an odd address.

Combinability

The "combinability" attribute of a segment is defined by the compiler, or by the user in assembly language. Compiler–generated segments are always "concat" segments. The assembler can also create "common" segments. All absolute segments are uncombinable. If two object modules define the same segment, then `llink`'s action depends upon the combinability attribute of the segment. The possibilities are:

1. **Concat Segment**. The segments are concatenated so as to preserve alignment, and the references are adjusted accordingly. The length of the output segment is roughly the sum of the lengths of the input segments. The alignment of the output segment is the maximum of the alignments of the input segments.
2. **Common Segment**. The object modules are combined by overlay. The length of the output segment is the length of the largest input segment.
3. **Uncombinable Segment**. The linking locator cannot combine pieces of an uncombinable segment and therefore emits an error message.

The combinability attributes of segments are displayed by the global symbol map utility.

Note: The fact that absolute segments are uncombinable implies that located object modules often cannot be used as input to subsequent links. This is because most modules define a chunk of the `idata` or `udata` segments, which become uncombinable after locating.

Here is a summary of the different segments in the InterTools system:

Segment	Use
`S_`*fname*	Code segment for a module whose first function is the function *fname*
`S_`*vname*	data segment for separate variable *vname*
`idata`	Initialized non–separate global data
`udata`	Uninitialized non–separate global data
`sdata`	String constants
`cdata`	See `const` qualified variables (see `-cs` compiler switch)
`libcode`	Assembly language library code
`init`	Initialization library routine (`__main`)
`init@0`	Initial values of PC and SSP at address 0
`BREAKSEG`	XDB breakpoint routine
`BREAK-SEG@16`	Breakpoint trap vector at address hex 10

4.5.2 Groups

A **group** is a named collection of data segments; `llink` must place the segments of a group within a contiguous 64K range of target memory.

The InterTools compiler generates a group named `data` which consists of the `idata` and `udata` segments. By default all global variables with explicit initial values are allocated in `idata`; those without explicit initial values are allocated in `udata`.

The linking locator creates a global symbol named `ldata`, whose value is the size of the data group (idata and udata segments). This may be useful to programs which dynamically allocate their global data area. Note that `ldata` is not a segment; it is a global symbol.

Note: The "origin" of a group smaller than 32K is the smallest address in any segment in that group. The origin of a group larger than 32K is 32K plus the smallest address in any segment in that group. This allows a program to take advantage of more efficient addressing modes by using positive and negative 16-bit offsets to address groups larger than 32K.

The compiler does *not* use the group concept with code. Rather, the compiler places all code resulting from a single compilation into one segment. All subroutine calls are long, that is, 32–bit addressing, but loop control is done with short branches. As a result, individual subroutines are limited to 32K of generated code, while there is no limit on the total amount of code from a series of compilations.

Example

Locate segments in a group.

Suppose you want to independently locate two segments which belong to a group, say, `seg1` and `seg2`, both of which belong to group "`group1`". The following locator commands in a locator command file will *not* work:

```
locate  ( seg1 : #F0000 );
locate  ( seg2 : #FFFF0 );
```

The problem is that the first locator command causes llink to also locate seg2, since it wants to ensure that both segments will fit into the same 64K byte range. The following locator command *will* work.

```
locate ( seg1 : #F0000 , seg2 : #FFFF0 );
```

The difference is that llink only locates group1 after it has finished the entire `locate` command.

4.5.3 Classes

A **class** is a named collection of segments that share a common logical attribute, such as being executable code or data. We often use the notational convention of bracketing a name in curly braces to indicate that it is a class name. This convention is also accepted within the locator command language.

Class names provide a convenient "handle" by which one can refer to a long list of segments without naming each one individually. For example, it is possible to use a single `locate` command to place all the segments of a given class into a given range of target memory. Unlike groups, classes impose no size limit.

People sometimes confuse classes with groups. Groups play an important role in code generation. They influence the compiler's strategy for addressing data. In contrast, classes play no role in code generation. The class name of a segment is best thought of as an abstract attribute of that segment; it is a descriptive comment describing the meaning or intended usage of that segment. Users can make up their own class names and assign them whatever significance they want.

Every segment belongs to some class, even if it is only the null class, "{}". All assembly language code and data have the null class. The compiler assigns class names by the following rules:

- All code segments have class {code}, unless otherwise specified via the -cc compiler switch.
- All segments containing non–separate data have class {data}.
- The separate data segments which are assigned class names by directives like:

```
#pragma sep_on class defclass
#pragma sep_on class defclass defclass2
#pragma separate myvar class defclass
```

  or by switches like:

```
-sc defclass
-sc defclass defclass2
```

  have class {*defclass*} if *defclass2* is not specified. If *defclass2* is specified, then they have class {*defclass*} if initialized data and {*defclass2*} if uninitialized data.
- Other local static separate data segments have class {stsep}.
- When *defclass* and/or *defclass2* are not specified, global separate data segments have class {isep} if they contain initialized data, and {usep} if they contain uninitialized data.
- All segments containing string constants have class {constant}.

Here is a summary of the different class names in the InterTools system:

Class	Use
{code}	Code
{data}	Non–separate global data
{constant}	String constants, also see `const` qualifier
{isep}	Default initialized separate data class
{usep}	Default uninitialized separate data class
{stsep}	Default static separate data class
{separate}	Class name for separate data when a user–specified segment name is supplied without a user–specified class name
{}	Null class — assembly language code and data

4.5.4 Relocation

Most segments are **relocatable** prior to locating, that is, they can be placed anywhere in target memory, independently of other segments. Address references in relocatable segments are represented symbolically, so they can be correctly replaced with an absolute address reference after location is complete.

`llink` maintains an image of the target machine memory and allocates space for the segments in the input module. This is an automatic process which can be partially or completely controlled through `locate` and `reserve` locator commands. After all user–provided locator commands have been processed, the default placement algorithm allocates memory to any remaining segments.

The assembler can define **absolute** segments that are assigned absolute target memory addresses at assembly time. To avoid overlapping segments, `llink` first locates each absolute segment at its indicated address.

Next, your `locate` commands are processed. Each named segment or class of segments is allocated in the indicated address range. If any of these segments belong to a group, then the other segments in that group are also allocated at that time.

Any remaining segments are allocated according to the default placement algorithm. The default allocation begins at location zero and traverses the segments in an unspecified order. If a segment belongs to a group, `llink` attempts to locate the whole group within a single 64K range. Segments are allocated with no gaps between them, except where gaps are needed to honor segment alignment.

Varying amounts of memory are present in target environments. The actual amount depends upon the particular configuration of the target machine. Use the `memory` locator command to define the actual memory configuration.

4.6 Compiler Library Organization

There are several libraries included in the InterTools distribution. These libraries are described in detail in the *Runtime Library* chapter in the *InterTools Reference Manual.*

The different libraries are intended for use in differing situations: generally you will be able to link with the same library each time. The following issues will determine your library choice:

- Hardware/Software Floating Point

  This choice only applies to the MC68020, MC68030, MC68EC030, and MC68040; not the MC68000. The `-h` switch directs the compiler to use MC68881, MC68040, or MC68060 floating point instructions. By default the compiler uses emulation routines in place of hardware floating point instructions. The software floating point library contains these routines; the hardware floating point library doesn't. Furthermore, floating point operations performed within the software floating point library itself use emulation routines, while corresponding operations within the hardware floating point library use hardware floating point (MC68881, MC68040, or MC68060) instructions.

Note: Code compiled **with** `-h` must be linked with a hardware floating point library. Except for the MC68040 and MC68060, code compiled **without** `-h` must be linked with a software floating point library.

- Long/Normal Integers

  The compiler has a switch, `-L` which directs the compiler to treat the **int** or **short** data type as 4 or 2 bytes long, respectively. This affects the library, because routines with **int** parameters expect 4 bytes of data if called from C code compiled with `-L` but only 2 bytes of data if called from C code compiled without `-L`.

Note: Code compiled **with** `-L` must be linked with a long library. Code compiled **without** `-L` must be linked with a normal library.

Note: The C++ compiler uses `-L` by default and therefore requires a long library.

- Floating point/No Floating point

  The "no–floats" library has been stripped of all floating point emulation routines. If your program uses **NO** floating point data, then a considerable reduction in size can be achieved by using this library.

- C++ Support

  If your application uses C++, an additional library must be linked in. The C++ library index files can be found in the `cpplib` directory, under `rtlibs`. The choice of index file depends on the target processor. The `cpp000.lib` library index file is used when compiling for the MC68000, MC68EC000, MC68008, MC68010, and MC68302 targets. Otherwise, the `cpp020.lib` library index file is used.

Note: Linking in a C++ library must always be done in addition to linking in a standard InterTools library.

The following tables summarize the libraries included in the InterTools distribution.

(The PC directories shown are the default directories used by the installation program. These may have been changed by your system administrator.)

MC68000, MC68EC000 and MC68008 Targets:

Directory	Index File	Integer Type	FP Included
`lib000\lib`	`lib000`	Normal	Yes
`lib000\lib`	`lib000.nf`	Normal	No
`lib000\lib`	`lib000.l`	Long	Yes
`lib000\lib`	`lib000.lnf`	Long	No

MC68010 Target:

Directory	Index File	Integer Type	FP Included
`lib010\lib`	`lib010`	Normal	Yes
`lib010\lib`	`lib010.nf`	Normal	No
`lib010\lib`	`lib010.l`	Long	Yes
`lib010\lib`	`lib010.lnf`	Long	No

All the M68000 family libraries follow the same pattern. Only the root name (`lib000` and `lib010` above) changes. Here are the root names for the other possible combinations:

Target Processor	Floating Point Package	Target Board	Library Directory
MC68000	SW	—	lib000
MC68EC000	SW	—	lib000
MC68008	SW	—	lib000
MC68010	SW	—	lib010
MC68020	SW	—	lib020s
MC68020	HW	—	lib020h
MC68EC020	SW	—	lib020s
MC68EC020	HW	—	lib020h
MC68030	SW	—	lib030s
MC68030	HW	—	lib030h
MC68EC030	SW	—	libe30s
MC68EC030	HW	—	libe30h
MC68040	on–chip f.p.	—	lib040
MC68EC040	SW	—	libe40
MC68302	SW	—	lib302
MC68302	SW	M68302ADS Development System using parallel I/O	lib302ap
MC68302	SW	M68302ADS Development System using trap I/O	lib302at
MC68330	SW	—	lib332
MC68331	SW	—	lib332
MC68332	SW	—	lib332
MC68060	HW	—	lib060h

Target Processor	Floating Point Package	Target Board	Library Directory
MC68340	SW	—	`lib340`
MC68340	SW	M68340BCC Development System	`lib340b`
MC68360	SW	—	`lib360`
MC68360	SW	M68360QUADS Development System	`lib360b`

4.7 Library Searches

`llink` does not begin searching for each external reference at the beginning of the list of libraries. Rather, it starts at the first library and continues searching until it cannot find the current external in the current library. Once it finds an external in a secondary library, it continues to search in that library until it cannot find an external there. It will eventually return to the first library, but only after searching all subsequent libraries.

This search pattern was designed to be efficient, but it has an important side effect. If an external is defined in more than one module, then `llink` **might not choose the one in the library named first.**

This presents no problems unless the same global is defined in more than one library member. If the user has multiple libraries which define the same global name, then there are two alternatives:

1. Delete members from the libraries (using the librarian) until they no longer overlap.
2. Link the system in stages, naming the desired libraries one by one at each step in the desired order.

Example

Search library files; write output on specified file.

Assume that *c:\itools\rtlibs* is the name of the PC directory or *itools/rtlibs* is the name of the Unix directory containing the compiler runtime libraries.

For the PC:

```
llink sort.ol -lo -L c:\itools\rtlibs\lib020s\lib\lib020s -o end.ln
```

For Unix hosts:

```
llink sort.ol -L itools/rtlibs/lib000/lib000 -lo -o end.ln
```

- Read `sort.ol`.
- Include necessary modules from the runtime library.
- Do not perform locate processing.
- Write relocatable linked module to `end.ln`.

4.8 Locator Commands

`llink` accepts commands from a command file. Commands define the layout of target memory and establish correspondence between external symbols and absolute addresses. The `-p` command line switch overrides any conflicting `SEGSIZE` command in the command file. The following table summarizes the available commands:

Command	Function
`DECLARE`	Define unresolved external symbol
`LOCATE`	Specify segment placement
`MEMORY`	Specify memory size
`RESERVE`	Reserve memory space
`SEGSIZE`	Pad segment sizes
`START`	Specify starting address

Example

Supply locator commands:

```
llink program.ln -c project.lc -o absolute.ab
```

- Read object module from file `program.ln`.
- Read locator commands from file `project.lc`.
- Write absolute module to file `absolute.ab`.

4.8.1 General Command Syntax

All locator commands consist of a command keyword followed by a left parenthesis "`(`", a sequence of operands, a right parenthesis "`)`", and a semicolon "`;`". For example:

```
LOCATE (init : #1000);
```

Insert blanks, tabs, or newlines freely to improve readability; they are only significant as separators of items in a list. The kinds of operands and the rules for forming them are discussed below.

4.8.2 Comments

Comments may be entered anywhere in a command file by prefixing them with two hyphens, "`--`". `llink` ignores all text between the dashes and the next newline.

4.8.3 Numbers

Numbers are used as target–machine addresses or as pad values. Numbers may be decimal or hexadecimal:

```
12345 is decimal
#A000 is hexadecimal
```

Hexadecimal numbers must be prefixed with the # character; the hex digits A to F may be entered in upper or lower case.

4.8.4 Keywords

Keywords may be entered in upper or lower case.

4.8.5 Address Ranges

An address range can be expressed in several forms:

low–address TO *high–address*
BEFORE *address*
AFTER *address*

The form BEFORE *address* is equivalent to 0 TO *address*; the form AFTER *address* is equivalent to *address* TO *end–of–memory.*

The low address is considered to be included in the range, but the high address is **not** considered to be included in the range. Thus the following two ranges do **not** intersect:

1000 TO 2000
2000 TO 3000

4.8.6 Names

Depending upon the context, names may be segment, class or global symbol names. Names are case–sensitive: `xy` is distinct from `XY`. Class names may be enclosed in curly braces, e.g., `{data}`, or tagged with the keyword `CLASS`, e.g., `CLASS (data)`, to distinguish them from conflicting segment or group names. An empty pair of curly braces, i.e., `{}`, indicates the null class.

4.8.7 Name List

A name list is a sequence of names separated by blanks.

4.9 Command Descriptions

The following pages describe the syntax of the individual locator commands. In cases where an optional repetition is allowed, a pair of square–brackets, `[` and `]`, enclose the repeatable pattern. Ellipses, "`. . .`", in this context refer to all of the previously specified operands.

Declare

Syntax:

```
DECLARE (name : address [,...]);
```

Function: *name* is the name of an unresolved external symbol. *address* is a 24–bit address for the external symbol.

The DECLARE command:

— Supplies an address for an unresolved external symbol.

— Resolves address references to the unresolved external symbol as if the external were located at the indicated address.

— Does not check to see if the address falls within any defined segment or even within the legal address range of the target machine.

External names from C programs must be those chosen via the compiler's naming conventions.

The DECLARE command can be used to repair references to external code routines, e.g., monitor routines in ROM.

The DECLARE command cannot always correctly resolve references to missing data items, because compiled code contains assumptions about how the missing variable will be addressed. In particular, the compiled code assumes that non–separate global data lie in group "data." References to missing **separate** data items CAN be repaired.

Example

```
DECLARE (_floppy_in : #FFE0,
   _floppy_out : #FFF0 );
```

In this example, `floppy_in()` and `floppy_out()` are external routines.

The names `floppy_in` and `floppy_out` are global symbol names, **not** segment names. The compiler prepends an underscore to the source name of a function when forming the global symbol name.

Example Catch calls to missing routines.

Let `gone` be a procedure which has not yet been coded. Suppose that calls to `gone` appear in code to be tested before `gone` is ready. If a call to `gone` is actually executed during testing, then the program will branch to location zero, with the usual undesired results.

Let `hero` be a procedure written to handle wild calls. `hero` may, for example, print a message and then return or cause a trap. The following locator command file specifies all calls to `gone` are to be re–directed to `hero`:

```
LOCATE ( hero : #1000 );
DECLARE ( gone : #1000 );
```

Locate

Syntax:

```
LOCATE (name–list : address–range [, ...]);
```

Function: In the previous syntax statement, *name–list* is a list of segment, group or class names. *address–range* defines the placement of named items.

The LOCATE command:

— Directs `llink` to locate a segment or a collection of segments in a specific region of target memory.

— After locating all the segments named in the *name–lists*, their groups (if any) are also located.

The list of names may be any combination of segments, groups and classes. Mention of a class is taken to mean the segments in that class which have not already been located.

Using the "address" form of *address–range* is equivalent to using the AFTER *address* form when more than a single segment–name is supplied in the name–list.

Note: Global symbol names may not be used in the name–list; use the corresponding segment name instead.

Example

```
LOCATE ( S_separate_var   : #3E00 );

LOCATE ( {code} {} {data}   : AFTER
#200 );

LOCATE ( CLASS (code)  : #200 TO
#1400,
        {} : #1400 TO #3000,
        idata   : #3000,
        udata   : #3800,
   CLASS (data)    : #3000 TO #C000);
```

Memory

Syntax:

```
MEMORY ( address ) ;
```

Function: In the previous syntax statement, *address* is the maximum target memory address.

The MEMORY command:

— Specifies the total amount of virtual memory that llink may allocate. The address given must be less than or equal to #FFFFFF.

— Allows specification of addresses larger or smaller than normally available for the target machine.

— Can be used when the processor allows bank memories.

The default maximum address depends upon the target processor.

The actual address specified is considered **not** to be available for any segment.

If `llink` attempts to locate something outside the MEMORY limitation, `llink` will tell you that you shouldn't locate something outside the addressing limitations of the processor. The MEMORY directive declares this limit.

Example `MEMORY ( #FFFFFFFF );`

This command means that you can allocate memory from 0 to n–1 where n = FFFFFFFF. n is the maximum amount of memory used in the environment.

Example Multiple address spaces.

There are several hardware "memory management" devices which allow a processor to access multiple address spaces, depending upon its state. For example, it is possible to access different physical memory, depending upon whether the memory access is a code or data fetch. This example is not intended to show how to interface with any particular device, but rather to offer some ideas which can be used to devise your own interface.

In an example for the MC68000, we form an absolute module in which two 24–bit virtual address spaces (one for code, the other for data) are mapped into a single 25–bit physical address space. Addresses 0 to #FFFFFF contain data; addresses #1000000 to #1FFFFFF contain code. The following locator commands expand memory and force all the code and data into the proper address ranges:

```
MEMORY   ( #2000000 );
LOCATE   ( {data} {isep} {usep} {stsep}
         {constant}     : BEFORE #1000000
);
LOCATE   ( {code} {}    : AFTER #1000000 )
```

The located entities represented here are:

Class	Contents
{data}	C global data
{constant}	C string constants
{isep}	Initialized C global separate data
{usep}	Uninitialized C global separate data
{stsep}	C static separate data
{code}	Code from C
{ }	Code from assembler routines in runtime library

This scheme works because the MC68000 processor only uses the low order 24 bits when it processes an address. Thus, a branch instruction whose target is address #1000234 will transfer control to #234 (in the code address space).

The following formatter switches can be used to extract the code and data images separately. See the *Formatter* chapter for more details.

```
-a 1000000 -w 1000000    Extracts code
only
-w 1000000               Extracts data
only
```

Reserve

Syntax:

```
RESERVE ( address-range [, ...] );
```

Function: In the above syntax statement, *address–range* represents the region of target memory to be avoided during locate.

The RESERVE command:

— Specifies areas of memory **not** to be allocated by `llink`.

— Can be used to avoid a region that contains a ROM monitor, or memory dedicated to mapped I/O.

— Can be used to designate "holes" in the range of memory addresses that may exist in a particular target system.

— May **not** designate a range of memory addresses containing segments.

— Should precede any LOCATE commands in the command file.

Note: The lower bound of address–range is included in the reserved space; the upper bound is not.

Example Allocation begins at #0800:

```
RESERVE ( #0000 TO #0800 );
```

Allocation is made from #1F00 through #2FFF and from #4000 through #6FFF:

```
RESERVE ( #0000 TO    #1F00,
          #3000 TO    #4000,
                AFTER #7000 );
```

Example Allocation is allowed from #1F00 to #3000, and from #4000 to #7000:

Segsize

Syntax:

```
SEGSIZE ([name:] number [%] [, ...]);
```

Function: In the above syntax statement, *name* is the name of the segment to be padded (optional); If name is omitted, the pad is applied to all segments. *number* `[%]` is the amount of padding.

The `SEGSIZE` command:

— Directs `llink` to reserve a specified amount of "growth room" at the end of a specified segment or all segments.

— Is more specific than the command line option because the user can supply segment names.

If the percent sign is absent, *number* represents a pad size in words. Each segment's length will be increased by *number* words regardless of its original length.

If the percent sign is present, *number* (*number* > 100) represents a percentage pad. Each segment's length will be increased to *number*/100 times its original size.

Example

SEGSIZE (counter : 300, control_blks : 150%);

SEGSIZE (120%);

Start

Syntax:

```
START ( location-option );
```

Function: In the above syntax statement, *location–option* represents the address of segment name. The START command:

— Specifies the address in the absolute output module where execution is to begin.

— May use a segment name, but not a global symbol name, to define the start address.

The assembler has an option to define the "start address". If this option is invoked, the assembler places the definition into a special ".start" record in the object module. A compiled program typically picks up its start address from the runtime library, which defines a starting point at the symbol *__main*. An assembler *main* routine can define its own starting point.

The effect of the START command is to set the value in the .start record in the absolute module. It can be used to override the start address supplied in the input module(s), or to specify a start address where none is supplied in the input module(s). If an absolute address is specified in the START command, the user must take care to ensure that whatever routine he intends to have control at system startup is actually located at the indicated address.

The starting address is copied into the download file by the formatter. Any effect this may have is determined by the target loader program. On some systems, downloading a file with a defined start address causes the program counter register to be set to that value. Systems which intend to begin execution after a cold start generally do not need to define a start address.

Example Start execution at origin of segment `S_my_main`:

```
START ( S_my_main );
```

4.10 Messages

A list of warning messages that can be emitted by the linking locator appears below.

Warnings

Class *class* of segment *segment* conflicts with new class *class1*

Command file *file–name* does not exist

Conflicting absolute addresses for segment *segment*

Date *date* in library doesn't match object module

Didn't get definition for register variable *variable*

Extra startup address ignored

Ignoring negative segment alignment reassignment

Ignoring negative segment alignment value

Ignoring new target *target–name*

Illegal expression operator

Illegal .id translator line

Input file switch *switch* ignored

Input modules listed on command line and in a file

Internal error in expression processing

Invalid module. id number *number*

Invalid module. id number *number* <=0

Invalid object module. Bad identifier: *identifier*

Keep symbol *symbol* not found in this link

No name with first occurrence of id *number*

No name with first occurrence of *symbol*

Segment *segment* defined as both common and non–common

Segment *segment* exceeds maximum size

Segment *segment* is already in a different group

Segment *segment* is uncombinable

START command overrides object module's

Suppress symbol *symbol* not found in this link

Symbol *symbol* has already been defined

Errors

Attempt to locate undefined object *object*

Can't open input module–list file *file–name*

Can't open library file *file–name*

Can't open output file *file–name*

Can't open temp file *file–name*. Writable directory required

Command line switch *switch* ignored; value required

Illegal .id *number* line.*number*

Incomplete module. No length associated with *segment*

No available memory for *object*

No input module names in module–list file

No input module names in module–list file *file–name*

Out of disk space for output file

Recursive definition of global symbol *symbol*

Redefinition of type of symbol *symbol*

Resulting segment size too big: *number* words

CHAPTER 5

Formatter

This chapter describes the operation and use of the two formatter utilities, `form` and `form695`. The chapter begins with a summary listing of the available options and continues with more detailed explanations of their usage and a list of error messages.

5.1 Introduction

Function: Format load modules for target system.

Invocation:
`form` [*prog*.`ab`] [*options*]

Input: Standard input or *prog*.`ab`

Output: *prog*.`hex` [*prog*.`xdb`] [*prog*.`asc`] or
prog.`X` [*prog*.`L`] [*prog*.`A`] [*prog*.`xdb`]

Function: Format load modules for IEEE–695 target system.

Invocation:
`form695` [*prog*.`ab`] [*options*]

Input: Standard input or *prog*.`ab`

Output: *prog*.`x`

5.2 Formatter Options: Summary

The formatter recognizes the following options:

Option	Function	See Page:
`-0`	(Zero) Display the version number of executable.	5–23
`-a` *bias*	**`form` only.** Specify address of window to be formatted.	5–16
`-b` *x y*	**`form` only.** Control PROM byte slicing.	5–17

Option	Function	See Page:
`-br`	**`form` only.** With `-f c`: Reverse byte ordering within the COFF file. **`form` only. Unix only.** With `-x`: Generate XDB symbol file on a Sun4 host that is compatible with MS–DOS hosted debuggers.	5–19 5–20
`-c`	**`form` only.** Suppress the prepending of an extra '_' (underscore) character to external symbol names.	5–19
`-d [only] [anycase]` [*sfmt*]	**`form` only.** Include symbolic records in formatted output. If `only` is specified, include only symbolic records. If `anycase` is specified, then lower case letters are preserved in Intel binary (omf86) format output. If present, *sfmt* identifies the symbolic record format.	5–8
`-d [abs]`	**`form695` only.** Include symbolic records in formatted output. If this option is used, the debug part of the IEEE–695 output file will be present.	5–12
`-e` [*seg1...*]	Exclude named segments from output. If no segments are named, exclude "udata."	5–23

Option	Function	See Page:
`-ec` [*class1...*]	Exclude named classes of segments from output. If no classes are named, the switch is ignored.	5–24
`-err` [*file*]	**PC only.** Write error messages to *file*.	5–24
`-err+` [*file*]	**PC only.** Append error messages to *file*.	5–24
`-f` *format*	**`form` only.** Specify the ASCII hex or binary output format.	5–13
`-i` [*seg1...*]	Include only named segments in output. If no segments are named, include no segments.	5–24
`-ic` [*class1...*]	Include only named classes of segments in output. If no classes are named, include no classes.	5–24
`-m` [*reclen*]	**`form` only.** Specify the maximum length record to be output by the formatter. If *reclen* is omitted, use largest possible record length.	5–24
`-n`	Allow an unlimited number of errors without aborting. By default the formatter aborts after 150 errors.	5–26
`-no`	**`form` only.** Do not hash symbol types when building the XDB symbol table.	5–20
`-o` [*ofn*]	Write output to file *ofn*. If *ofn* is omitted, write to *prog*`.hex` (`form`) or *prog*`.x` (`form695`).	5–26

Option	Function	See Page:
`-st` *target*	**`form695` only.** Use *target* in the module begin (`MB`) `Id1` field instead of the target string found in the input file.	5–16
`t` *sspec1* [*sspec2...*]	**`form` only. Unix only.** Control symbol table sizes used in making the XDB symbol table.	5–21
`-t m` *size*	**`form` only. PC only.** Control the size of an internal mapping table used in making the XDB symbol table.	5–22
`-w` *size*	**`form` only.** Specify size of window to be formatted.	5–18
`-x` [*xfn*]	**`form` only.** Write XDB symbol table to file *xfn*. If *xfn* is omitted, write to *prog* `.xdb`.	5–23

5.3 Usage

5.3.1 `form`

The formatter reads an absolute object module produced by the linking locator and converts all or part of it into one of the industry standard formats, usually an ASCII hex format. These formats provide for loading of object text, i.e., code and data, into the memory of the target processor via a simple loader program. The formats may also be input to PROM burners, hardware devices which can program read–only memories. Many formats are supported; see the *Format Options* section below for a detailed listing.

When the formatted output is intended to be used as input to a PROM burner, you may want to produce several formatted files froma single input file, one for each PROM. It is possible to extract a range of target addresses, and/or select slices of memory, for example, every other byte. See the *PROM Options* section below for more details.

The formatter also produces many different symbol formats. There are two symbol generation switches available. The `-d` switch creates symbol records that may be used by a variety of emulators or downloading programs. The `-x` switch produces symbol files for use with CrossView Pro.

For the formatter to be able to include line number symbols, local static symbols, or type information in both the symbolic records (when supported by the specific format) and the XDB symbol files, compilations or assemblies must be done using the `-d` switch.

5.3.2 `form695`

The formatter reads an absolute object module produced by the linking locator and converts all or part of it into IEEE–695 object module format. The format provides for loading of object text, i.e., code and data, into the memory of the target processor via an IEEE–695 loader program.

The formatter also produces symbol information (IEEE–695 debug information part). The `-d` switch creates symbol records that may be used by an emulator or other hardware and software that accepts IEEE–695 input. For the formatter to be able to include line number symbols, local static symbols, or type information in the IEEE–695 output file the `-d` switch must also be used when compiling source files.

The CrossView Pro debugger uses an IEEE–695 file as input.

5.4 Formatter Options: Detailed Descriptions

This section describes the formatter options in greater detail and includes examples of their use.

5.4.1 Format Options

`-d [only] [anycase] [`*sfmt*`]`

`form` only. The `-d` switch controls the generation of symbolic records. Symbolic records, if generated, are placed into the formatted output file or into a separate file. Do not confuse these symbol records with the XDB symbol table file. If `only` is specified, ordinary target memory loading records are **not** emitted.

If `anycase` is specified, then lower case letters are preserved in Intel binary (omf86) format output.

The effect of the `-d` switch is dependent on the file format that has been selected with the `-f` switch. Some formats; Extended Tekhex, Binary Tekhex, COFF, and HP64000 for Unix hosts; Binary Tekhex, and COFF for the PC; have their own standard form for symbolic information. In these cases, *sfmt* and `only` must **not** be specified. Here is a brief summary of the symbolic information each format produces. For more detailed information on the formats in general, please refer to the *Object Module Formats* appendix.

Industry Standard Symbol Formats:

- **Extended Tekhex**

With the Extended Tekhex format, symbolic information is produced for global symbols only.

- **Binary Tekhex**

With the Binary Tekhex format, symbolic information is produced for global symbols only. Symbol names longer than 16 characters are truncated.

- **COFF**

With the COFF format, symbolic information is produced for global and local symbols, line number symbols and type definitions.

- **HP64000 (Unix only)**

With the HP64000 format, symbolic information is produced for global symbols, local static symbols, and line number symbols.

Non–Industry Standard Symbol Formats:

Other download formats have no industry standard format for symbolic information. For these formats, use the *sfmt* to select one of the "almost standard" symbol formats:

sfmt Value	Formats	Meaning
nwis	i, m, xm, pm, z, t, et	MicroCASE (Northwest Instruments Systems) ASCII Format
pe	map	P+E Microcomputer Systems
zax	i, m, xm, pm, z	ZAX Corporation ZICE–compatible symbols

- **nwis**

With the nwis symbol format, symbolic information is provided for global symbols, local static symbols, and line number symbols. Additionally, all of the type information useful to the MicroCASE SoftAnalyst is provided. If MicroCASE ASCII (NWIS ASCII) format is selected, symbol information will be put into a separate file that is suitable for use with the MicroCASE SoftAnalyst. This file is named *prog*.asc. If **only** is specified for NWIS format, then only the file *prog* . asc is produced.

- **pe**

With the pe symbol format, a generated map file can be used with the P+E low-level debugger and toolset normally used on CPU 32 targets.

- **zax**

With the zax symbol format, symbolic information is provided for global symbols, local static symbols, and line number symbols.

`-d` [*abs*]

`form695` only. The `-d` switch controls the generation of symbolic records. Symbolic records, if generated, are placed into the formatted output file. The debug part of the IEEE–695 output file will be omitted unless this switch is present. If *abs* is used, all global variables that are group data relative (not separate data) are given absolute addresses in the debug information part of the output file. This may be used to examine global variables even when the static base register has not been initialized.

Example

To generate an IEEE–695 file and include symbolic records, type:

```
form695 test.ab -d
```

- Read input from file `test.ab`.
- Write output which includes the debug part to the file `test.x`.

To generate an IEEE–695 file for use with CrossView Pro, type:

```
form695 test.ab -d abs -o test.abs
```

- Read input from file `test.ab`.
- Write output which includes the debug part to the file `test.abs`
- Give all global variables absolute addresses in the debug part.

`-f` *format*

`form` only. Select the output format; the default format is Packed Motorola (`pm`). For information on the formats, see the *Object Module Formats* appendix. The available format options are:

`bt`	Binary Tekhex.
`c`	COFF (Common Object File Format).
`c1`	COFF1 format. Identical to COFF, except that line numbers start at 1, and thus directly correspond to the program's line numbers. See the *Object Module Formats* appendix.
`et`	Extended Tekhex.
`hp`	**Unix only.** Hewlett Packard HP64000 (absolute, linker symbol, and assembler symbol files).
`i`	Intel ASCII hex.
`m`	Motorola (S records). Data in S1 records.
`pm`	Packed Motorola (S records). Data in S1, S2, or S3 records, where the record type is chosen by the number of address bytes.
`s37`	S37 Motorola (S records). Data is in S3 records.

NOTE: This format does not provide an S0 header record.

`t`	Tektronix ASCII hex (Tekhex).
`xm`	Extended Motorola (S records). Data in S2 records.
`z`	Z80SBC format. This is identical to Intel ASCII format, except the start address appears in the end record instead of in a special record.

Example

To use Extended Tekhex format and generate a debugger symbol file, type:

```
form test.ab -f et -x
```

- Read input from file `test.ab`.
- Output is in Extended Tektronix hex format.
- Write output to file `test.hex`.
- Write XDB symbol file `test.xdb`.

Example

To include symbolic hex records in output, type:

```
form myprog.ab -f et -d -o out.hex
```

- Read input from file `myprog.ab`.
- Output is in Extended Tektronix hex format.
- Include load module and symbolic debugging information in `out.hex`.
- Write output to file `out.hex`.

Example

To produce NWIS ASCII symbol information, type:

```
form myprog.ab -f pm -d nwis -o out.hex
```

- Read input from file `myprog.ab`.
- Output is in Packed Motorola S–record format.
- Put NWIS ASCII symbol information in `myprog.asc`.
- Write output to file `out.hex`.

Example

To produce only ZAX ZICE symbolic information in output, type:

```
form myprog.ab -f i -d only zax -o myprog.zax
```

- Read input from file `myprog.ab`.
- Put only symbolic debugging information in `myprog.zax`
- No target–loading hex bytes are in output file.
- Write output to file `myprog.zax`.

`-st` *target* — **`form695` only.** Use *target* in the module begin (MB) Id1 field instead of the target string found in the input file. Some emulators need the exact processor type instead of the more general name. For example, `-st 68302` would cause the default 68000 target type to be replaced with 68302.

5.4.2 PROM Options

`-a` *bias* — **`form` only.** The *bias* is a value to be subtracted from each target load address of the output hex file. The *bias* is an unsigned hex value, with up to 8 hex digits. This feature may be used, for example, to let a PROM programmer load a hex module into its location 0, which is actually located at the target address *bias*. Object text whose address is less than *bias* will not be emitted.

`-b x y`

`form` only. This switch supports byte slicing, which is useful when burning interleaved PROMs. Interleaved PROMs are used in hardware designs where the low order address bit(s) select different PROM chips.

Two decimal numbers are required, *x* and *y*, the second of which, *y*, must be a power of 2 (*y* is 2**n*). The switch causes the output to contain only every *y*th byte, with address bits shifted right by *n* bits, i.e., the output address is the input address (minus the bias, if any) divided by *y*. The input addresses chosen for output are those congruent to *y* mod *x*. See the table below for some examples:

Switch	Meaning
`-b 0 2`	Output every second byte, even addresses.
`-b 1 2`	Output every second byte, odd addresses.
`-b 3 4`	Output every fourth byte, address congruent to 3 mod 4.

-w *size*

form only. Defines the size of a window of object text from the input object file to be emitted to the output hex file. This option is usually used in conjunction with the `-a` option. *size* is an unsigned hex value, with up to 8 digits. The addresses emitted into the output file range from 0 (after any biasing) to *size* minus 1. If byte slicing is not done, this is the maximum size in bytes of the object text in the output hex file. If byte–slicing is done, the number of bytes of object text in the output hex file will be divided by the number of slices.

Example

To format a range of addresses, type:

```
form prog.ab -w 8000 -a 38000
```

- Output addresses range from 0 to 7FFF.
- The output hex file can contain up to 8000 hex bytes.
- Ignore input text outside addresses 38000 to 3FFFF.

Example

To use PROM byte slicing with a range of addresses, type:

```
form prog.ab -w 8000 -a 38000 -b 0 2
```

- Output addresses range from 0 to 3FFF.
- The output hex file can contain up to 4000 hex bytes.
- Ignore input text outside addresses 38000 to 3FFFF.
- Ignore input text at odd addresses.

See the *Downloading* application note for further information about the PROM options.

5.4.3 COFF Format Options

`-br` **`form` only.** When used in conjunction with the `-f c` switch, this switch will reverse the byte ordering within the COFF file. By default, the byte ordering is chosen with respect to the target processor. The section data is not modified.

`-c` **`form` only.** By default, `form` prepends an extra underscore to all external names entered into the COFF symbol file. Use of this switch will cause names to be entered without alteration. This switch is useful for COFF implementations that do not expect external names to have double underscores.

5.4.4 CrossView Pro Options

`-br` **`form` only. Unix only.** When used in conjunction with the `-x` switch, this switch will cause the Sun4 hosted formatter to generate an XDB symbol file that is compatible with a PC hosted debugger. This is useful when formatting very large applications that require a workstation and debugging using an MS–DOS hosted XDB.

`-no` **`form` only.** Do not hash symbol types when building the XDB table. By default, symbol types are hashed to optimize the size of the XDB symbol table. Using this switch may increase the speed of formatting since for some large input files hashing may result in slightly slower processing.

–t *sspec1* [*sspec2...*] **`form` only. Unix only.** Control symbol table sizes used in making the XDB symbol table.

Note: The `-t` switch has no effect without the `-x` switch.

This option allows the user to increase the size of certain internal tables used by the formatter when creating the XDB symbol file. If table limits are exceeded, the formatter will abort and emit an error message which describes how to use this switch and how many of each kind of object the formatter had encountered so far. Requesting uniformly larger tables with `-t L` will suffice for all but extremely large applications.

It is possible to control the size of each table individually, but this is only necessary if the formatter runs out of table space even with `-t L` or if the formatter runs out of memory. In that case, the switch operand can be specified as a list of pairs consisting of a letter (`p`, `l`, `d`, `s`, `m`, or `t`) followed by a size. The letters refer to the tables containing, respectively, procedures, line numbers, symbols, strings, mappings, and type hashing. For example, `-t p 2000` sets the size of the table of procedures to 2000.

If the formatter runs out of memory, decrease some table sizes. For example, the `t` switch operand sets the size of a buffer used for hashing symbol types. If not explicitly set, this value may be larger than necessary, based on the value given for the `d` switch. In this case, specifying a small `t` value (e.g., `-t d 5000 t 1000`) may improve performance. If the formatter runs out of room for a particular item, increase the size of that particular table. If all such attempts fail, then some portion of the system must be recompiled without symbol information.

Note: The `-t` switch has no effect without the `-x` switch.

`-t m` *size*

`form` only. PC only. Control the size of an internal mapping table used in making the XDB symbol table. If the limit of the mapping table is exceeded, the formatter will abort and emit an error message which describes how to use this switch. For example `-t m 7500` sets the map number limit to 7500.

Note: The `-t` switch has no effect without the `-x` switch.

`-x` [*xfn*] **`form` only.** Write a symbol table to file *xfn*; default is *prog*`.xdb`. This symbol table is appropriate for use by the InterTools source–level debugger, XDB. This symbol file should not be confused with any symbol information produced by the `-d` switch.

Note: If the user supplies xfn, it must have the `.xdb` filename suffix or it cannot be accessed by XDB.

5.4.5 Miscellaneous Options

`-0` (Zero) Display the version number of executable (for technical support use).

`-e` [*seg1...*] The `-e` switch excludes named segments from the hex or IEEE–695 output. This switch may be used, for example, to download only changed segments or to create a hex file of only ROM segments for a PROM–burning procedure.

When no segments are named, the formatter excludes the "udata" segment. This is appropriate if the user is sure the compiler's convention of default initialization to zero is either not needed or already ensured by some other means. For example, if the hex or IEEE–695 file is to be loaded to pre–zeroed memory, there is no need to format a segment containing only zeros. The default is to include all segments.

`-ec` [*class1...*]

The `-ec` switch excludes named classes of segments from the hex or IEEE–695 output. This switch may be used, for example, to exclude `#pragma separate` items. When no classes are named, the formatter ignores the switch.

`-err` [*file*]

PC only. Write error messages to file *file*. If *file* does not exist, it will be created. If *file* does exist, it will be overwritten. If *file* is omitted, then error output will be redirected to standard output.

`-err+` [*file*]

PC only. Just like `-err`, except output will be appended if *file* exists.

`-i` [*seg1...*]

The `-i` switch includes only named segments in the hex or IEEE–695 output. This switch may be used, for example, to separate compilations (or segments) that are patches to previous downloads.

When the `-i` switch is specified, but no segments are named, the formatter includes no segments.

`-ic` [*class1...*]

The `-ic` switch includes only named classes of segments in the hex or IEEE–695 output. This switch may be used, for example, to include only `#pragma separate` items. When the `-ic` switch is specified, but no classes are named, the formatter includes no classes of segments.

`-m` [*reclen*]

`form` only. This switch specifies the maximum length record to be output by the formatter. If *reclen* is supplied, it must not be less than 35

nor greater than 255. If *reclen* is not specified, the largest upper limit (255) is used. If this switch is not specified, a default is chosen depending upon the hex format.

The following table lists the defaults for the PC:

Lengths	Format
42	m and xm
80	pm
72	t
255	bt
46	et
80	i and z
255	omf86 and bi

The following table lists the defaults for Unix hosts:

Length	Format
42	m and xm
80	pm
72	t
255	bt
80	i and z
255	hp
255	omf86 and bi

`-n`	Allow an unlimited number of errors without aborting. By default the formatter aborts after 150 errors.
`-o` [*ofn*]	Write output to file *ofn*. If *ofn* is omitted, write output to *prog*.`hex` (`form`) or *prog*.`x` (`form695`). If `-o` is omitted, write output to *prog*.`hex` (`form`) or *prog*.`x` (`form695`).

Example

To exclude uninitialized pragma (option) separate items, type:

```
form prog.ab -f i -ec usep -o no_usep.hex
```

- Read input from file `prog.ab`.
- Output is in Intel ASCII hex format.
- Exclude all uninitialized `#pragma separate` segments (located in class `usep`) from output.
- Write output to file `no_usep.hex`.

5.5 IEEE–695 Formatter Limitations

The IEEE–695 formatter (`form695`) expects the input file to be an absolute located file generated by the InterTools `llink` linker/locator. Relocatable and unresolved symbols are not supported. The known limitations, restrictions, and problems are described below.

Register mask information is not generated for procedures. This means that variables packed to registers by the optimizer will not have the

correct displayed value in a debugger if the scope is not the current procedure. A workaround is to use the `-no` and `-nl` compiler switches instead of `-do` when compiling source code (in this case the optimizer will not be run at all and no user variables will be packed to registers).

5.6 Messages

A list of error messages that can be emitted by the formatter appears below.

Warnings

Binary Intel format not appropriate for this target processor.

BYTESLICE(2) not 2**n, using *value* instead.

Illegal value for format switch *val.*

Invalid bias format.

Invalid window size specified. *which* value of *value* used for *switch* command line switch.

No biasing done. Invalid bias value.

No biasing done. Invalid window size format.

Regular symbolic debugging info not available with this format.

Symbol name(s) truncated to 15 char maximum for BININTEL format.

Symbols not available with bytes slicing, bias or window options.

Using Intel format.

Warning: unknown storage class for symbol.

Warning: unsupported target type using m68000.

Zax compatible symbolic debugging info not available with this format.

Errors

Illegal register variable, *variable*.

Invalid object module: bad tag *char*.

Invalid object module: Token *token*, syntax error.

String too long (>127 bytes) in *emit string* — truncated.

Target value of *value* does not fit in *value*–bit

Target value refers to undefined symbol *symbol* or non–constant expression.

Unresolved reference to *symbol*

[Un]signed field. Can be caused by out of bounds reference, segment overflow, or invalid .ab file.

Fatal Errors

Fatal error: module missing segment: *module name.*

If appropriate, reformat with extended Motorola format.

If appropriate, reformat with Extended Tekhex format.

If appropriate, reformat with packed Motorola format.

Illegal value for format switch: *string*.

Invalid module: missing tag.

Non–printing character, hex (*val*).

Target address *val*, 16 bits, too large for this format.

Target address *val*, >24 bits, too large for this format.

Too many errors; execution abandoned.

Unable to create linker symbol file: *file*.

Unable to open assembler symbol file: *file*.

Unable to open input file: *file*.

Unable to open output file: *file*.

Unlocated object module.

CHAPTER 6

Other Utilities

This chapter describes the following additional utilities:

- Librarian
- Global Symbol Mapper
- Symbol List Utility
- Object Size List Utility

6.1 Librarian

Function: Manage object module library.

Invocation:

`libr` [*obj1...*] `-L` *prog.lib* [*options*]

Input: Object module library index file and object modules: *prog*`.lib` [*prog*`.obj1...`]

Output: New or updated *prog*`.lib` [*prog*`.lis`]

6.1.1 Librarian Options: Summary

The librarian recognizes the following options:

Option	Function	See Page:
`-0`	(Zero) Display the version number of the executable.	6–12
`-a` *obj1* [*obj2...*]	Add named object module(s) to library.	6–7
`-af` *afn*	Add object modules in file *afn* to library.	6–8
`-b` [*sym1...*]	List object files that define the named symbols.	6–11
`-c`	Check and report on header/index consistency.	6–11

Option	Function	See Page:
`-d` *obj1* [*obj2...*]	Delete named object module(s) from library.	6–8
`-df` *dfn*	Delete object modules in file *dfn* from library.	6–8
`-e`	Suppress updating of library index file if any warning messages occur.	6–8
`-err` [*file*]	**PC only.** Write error messages to file.	6–12
`-err+` [*file*]	**PC only.** Append error messages to file.	6–12
`-i` [*obj1...*]	List index header of named object modules.	6–11
`-i -b`	List the entire library.	6–11
`-i` *obj1* [*obj2...*] `-b`	List symbols of named object modules.	6–11
`-i` *obj1* [*obj2...*] `-b` *sym1* [*sym2...*]	List specified symbols in named object modules.	6–11
`-l` [*lfn*]	Write listing to *lfn*. If *lfn* is omitted, write to *prog*`.lis`.	6–11
`-L` *lib*	Specify the name of the library to be created, modified or listed.	6–7
`-n`	Suppress segment and group names in index.	6–8
`-rf` *rfn*	Replace object modules in file *rfn* in library.	6–8

Option	Function	See Page:
-u	Update all object modules in library.	6–9
-v	Report librarian actions as performed.	6–12

6.1.2 Usage

The librarian creates, maintains, and selectively lists library index files.

A library index file is a text file defining an indexing structure describing a collection of object modules. It consists of a series of index entries, one for each object module.

The library **does not contain the object modules themselves**, only their filenames and information extracted from them. Each index contains the following data:

1. A header containing:

- A full or relative pathname.
- A date/time stamp.

2. A list of global symbols defined in the object module.

This information is extracted from an object module and formatted as a library index when a module is "added" to the library, or when its index is "updated."

Librarian input is taken from a library and/or object modules named on the command line or in options. The object modules named on the command line or in a file are added to the library.

If you specify object modules on the command line but do not specify an add or delete option, the librarian will either replace or add the object modules to the named library, depending on whether the object modules are already in the named library.

Note: If the library already contains an index for a module, the index is updated only if the module is newer than the date stamped in the index.

The file containing names of the object modules to add, delete, or replace **must** be in a specific format of one object module name per line.

When an object module has been created, deleted or modified, each library containing that module **must** be updated. Otherwise, the library indexes may contain incorrect information about the global symbol names needed during the linking process.

The linking locator can search one or more libraries for modules that resolve references to external symbols. If the file name in the library is not a full pathname, the linking locator searches for the module in the directory containing the index file. This allows you to construct portable libraries. If the library index file and the object modules reside in a common directory, you may move or rename the directory without disturbing the functioning of the library. See the *Linking Locator* chapter for a more detailed explanation of the use of library index files.

Example

Catalog an object module with all defaults:

```
libr prog.ol -L project.lib
```

- Add `prog.ol` to library `project.lib`.
- If a newer version of `prog.ol` is already in `project.lib`, a warning is issued and `project.lib` is unchanged.
- If an older version of `prog.ol` is already in the library, its index is replaced.
- No listing is generated.

6.1.3 Librarian Options: Detailed Descriptions

This section describes the librarian options in more detail and provides examples of their use.

Library Option

`-L` *lib* — Specify the library to be created, modified or listed. If the library does not exist it will be created. This switch is required.

Command Options

`-a` *obj1* [*obj2...*] — Add new indexes for the named object modules to *prog.lib*.

`-af` *afn*	Add object modules in file *afn* to *prog.lib*. Object module names are either taken from the command line or from the named file, but not both. At least one object module name is required if this option is used. If an object module is already catalogued, its index is not replaced and a warning is issued.
`-d` *obj1* [*obj2...*]	Delete the indexes for the named object modules from *prog.lib*.
`-df` *dfn*	Delete object modules in file *dfn* from *prog.lib*. Object module names are either taken from the command line or from the named file, but not both. At least one object module name is required.
`-e`	Suppress updating of the library if any warning messages occur.
`-n`	Suppress segment and group names in the index. This makes a smaller library index, thus speeding up the library search. It is appropriate for compiler–generated object modules and the runtime library. This is safe because the compiler never generates external references to segment or group names. It is **not** appropriate for those user assembler modules whose segment names are used as external names in other modules.
`-rf` *rfn*	Object module names are taken from file *rfn*. Replace the indexes for the named object modules in *prog.lib*.

`-u` Update all indexes. This option is a global consistency check and replace operation. It is performed on the entire library. The consistency check compares all date/time stamps in the respective files. If any pair of date stamps does not match, the library index is updated. The librarian issues a warning if no file is found for a library index.

Example

Add new index for object module:

```
libr -n -L project2.lib -a myprog.ol
```

- Add index for `myprog.ol` to library `project2.lib`.
- The `-n` switch tells the librarian to strip off segment and group names. This makes for a more compact library, but is only appropriate for compiler generated object modules.
- If `myprog.ol` is already in the library, a warning is issued and the library is unchanged.
- No listing is generated.

Example

Create library from list of names in a file:

```
libr -L project6.lib -af addmods
```

- The file `addmods` contains:

  ```
  hello.ol
  sieve.ol
  ```

- Add `hello.ol` and `sieve.ol` to library `project6.lib`.

Example

Delete object modules from library:

```
libr -L project3.lib -d gjh.ol nph.ol
```

- Modify library index file `project3.lib`.
- Delete indexes for `gjh.ol` and `nph.ol`.
- The object modules themselves are **not** deleted.

Example

Update library:

```
libr -L project5.lib -u
```

- Access library index file `project5.lib`.
- Update all indexes to most recent versions.

Listing Options

`-b` [*sym1...*] — List named global symbols. If no symbols are specified, list all symbol entries.

`-c` — Check and report on the consistency of all entries in the library.

No changes are made to the library. Error messages are issued when:

1. An index in the library has no corresponding object module.

2. The date/time stamp in the library index does not agree with the date/time stamp in the object module, i.e., the library index does not correspond to the current object module.

`-i` [*obj1...*] — List the index headers of the specified object modules. If no object modules are specified, list all index headers.

`-l` [*lfn*] — Write listing output to file *lfn*. If *lfn* is omitted, write to file *prog*.`lis`. The default is to write the listing to standard output.

Combined Listing Options

`-i -b` — List the entire library.

`-i` *obj1* [*obj2...*] `-b` — List all global symbols in the named object modules.

`-i` *obj1* [*obj2...*] `-b` *sym1* [*sym2...*] — List the specified symbols in the named object modules.

Example

List library:

```
libr -L project4.lib -a hello.ol sieve.ol -i -b -l
```

- Add `hello.ol` and `sieve.ol` to library `project4.lib`.
- List all symbols in `project4.lib`.
- Write output to `project4.lis`.

Listing of `project4.lis`:

```
hello.ol          "Dec 31 1996  11:53:38"
   data
   idata
   udata
   sdata
   _i
   S_main
   _main
sieve.ol          "Jan 18 1996  11:33:25"
   data
   idata
   udata
   sdata
   _flags
   _main
   S_main
```

Miscellaneous Options

`-0`	(Zero) Display the version number of executable (for technical support use).
`-err` [*file*]	**PC only.** Write error messages to file.
`-err+` [*file*]	**PC only.** Append error messages to file.
`-v`	Report librarian actions as performed.

6.2 Global Symbol Mapper

Function: List global symbols and segments.

Invocation:

`gsmap` [*obj1...*] [*options*]

Input: Standard input or [*obj1...*]

Output: Standard output or *obj1*.`map`

6.2.1 Global Symbol Mapper Options: Summary

The global symbol mapper recognizes the following options:

Option	Function	See Page:
`-0`	(Zero) Display the version number of the executable.	6–19
`-a`	Print symbols in alphabetical order.	6–19
`-an`	Print symbols in alphabetical order and segments in address order.	6–19
`-err` [*file*]	**PC only.** Write error messages to file.	6–19
`-err+` [*file*]	**PC only.** Append error messages to file.	6–19
`-n`	Print symbols in address order.	6–19
`-na`	Print symbols in address order and segments in alphabetical order.	6–19
`-o` [*ofn*]	Write output to file *ofn*. If *ofn* is omitted, write to *obj*`.map`. By default, `gsmap` writes to standard output.	6–17
`-P` [*lines*]	Set lines–per–page to *lines*. If *lines* is omitted, suppress pagination.	6–17
`-s`	Omit listing of externals and globals; list segments only.	6–17
`-z`	Exclude empty segments from listing.	6–17

6.2.2 Usage

The global symbol mapper (`gsmap`) displays symbolic information from an object module. This utility can be used before or after linking or locating. `gsmap` lists external names and the definitions of global symbols. The `SEGMENT` section of the `gsmap` listing shows absolute address, length, class and alignment for each segment.

The `gsmap` listing can be more easily understood once you know the compiler's naming conventions. See the *Compiler Naming Conventions* appendix for an explanation.

Example

Produce alphabetic global symbol listing:

```
gsmap hello.o1
```

- Reads object module `hello.o1`.
- Writes listing to standard output.

Given the following C program:

```
main() {
    printf ("Hello, world.\n");
}
```

The alphabetic map listing produced is:

```
Symbol Map for hello.ol Feb 10 1997 11:30:41 Page 1

Translator: Unknown
Target    : 68000

Externals

__main
_printf

Global Address

_main  E1

Group     Size Limit       Align     Member       Segmen
ts

data      00ffff (65535)   hword     udata     idata

Segment       Address   Length Class      Align  Combine

S_main    000010 (16)   {code}    hword  concat
idata     000000 (0)    {data}    hword  concat
sdata     00000e (14)   {constant}hword  concat
udata     000000 (0)    {data}    hword  concat

Statistics

Segments  : 4
Externals : 2
Globals     : 1
Groups    : 1
Code Size : 00000010 (16)
```

6.2.3 Global Symbol Mapper Options: Detailed Descriptions

This section describes the global symbol mapper options in more detail and provides examples of their use.

Listing Options

`-o` [*ofn*]	This switch specifies the name of the output file. If *ofn* is omitted, the output is written to file *obj*.`map`, where *obj* is the root of the first mappable file. If input is taken from stdin, the default output file name is `stdin.map`.
`-P` [*lines*]	Set the number of lines per listing page to *lines*. If *lines* is omitted, suppress listing pagination. The default is to emit a new title heading every 60 lines.
`-s`	Omit listing of externals and globals; list segments only.
`-z`	Omit empty segments from output listing.

Example

Set the number of lines per page in the output listing:

```
gsmap prog.ln -P 22 -o prog.out
```

- Reads linked object module `prog.ln`.
- Sets the number of lines per listing page to 22.
- Writes listing to `prog.out`.

Example

Exclude empty segments :

```
gsmap test1.ab test2.ab test3.ab -z -o
```

- Reads input from `test1.ab`, `test2.ab` and `test3.ab`.
- Produces a symbol map for each input file.
- Excludes empty segments from each symbol map.
- Writes listing containing all three symbol maps to `test1.map`.

Sorting Options

`-a`	Print symbols in alphabetical order.
`-an`	Print symbols in alphabetical order and segments in address order.
`-n`	Print symbols in address order.
`-na`	Print symbols in address order and segments in alphabetical order.

If both the `-a` and `-n` switches are specified, segments and global symbols are listed twice, once in each order. The default is to print in alphabetical order.

Miscellaneous Options

`-0`	(Zero) Display the version number of the executable (for technical support use).
`-err` [*file*]	**PC only.** Write error messages to file.
`-err+` [*file*]	**PC only.** Append error messages to file.

6.2.4 Messages

unable to open input file *file*

syntax error: token *token* line *line*

6.3 Symbol List Utility

Function: Display symbolic information from object module.

Invocation:

`symlist` [*prog*`.[ol | ln | ab]`] [*options*]

Input: Standard input or [*obj1*...]

Output: Standard output or *prog*`.sml`

6.3.1 Symbol List Utility Options: Summary

The symbol list utility recognizes the following options:

Option	Function	See Page:
`-0`	(Zero) Display the version number of the executable.	6–23
`-err` [*file*]	**PC only.** Write error messages to *file*.	6–23
`-err+` [*file*]	**PC only.** Append error messages to *file*.	6–23
`-o` [*ofn*]	Write output to file *ofn*. If *ofn* is omitted, write to *prog* `.sml`.	6–23

6.3.2 Usage

The symbol list utility, `symlist`, produces a listing of all symbols, global and local, along with target locations for source lines of input code. The input may be any combination of unlinked object modules, linked object modules, or absolute object modules. If no object modules are named, then standard input is read.

Note: Only object modules which were compiled or assembled with the `-d` option will include symbolic information which can be listed.

Example

Produce symbol list output file:

```
symlist prog1.ln prog2.ln prog3.ln -o
```

- Reads object modules `prog1.ln`, `prog2.ln`, and `prog3.ln`.
- Writes listing to `prog1.sml`.

6.3.3 Symbol List Utility Options: Detailed Descriptions

This section describes the symbol list utility options in more detail.

`-0`	(Zero) Display the version number of executable. For technical support use.
`-err` [*file*]	**PC only.** Write error messages to *file*.
`-err+` [*file*]	**PC only.** Append error messages to *file*.
`-o` [*ofn*]	This switch specifies the name of the output file. If *ofn* is omitted, the output is written to file *obj*.`sml`, where *obj* is the root of the first file that can be successfully run through `symlist`. If input is taken from standard input, the default output file name is `stdin.sml`.

6.3.4 The Symbol Table Listing

The `symlist` listing can be divided into two main parts. The first part contains symbol information for each compilation unit. It begins with the header line:

```
INDEX NAME      SCOPE CLASS, ATTRIBUTES
```

Following the header line is a list of executable line numbers and code addresses for linked or located object modules, with relative code addresses for linked modules, and absolute code addresses for located modules.

Next is a listing of all symbols and their attributes. In each entry, the `INDEX` is a number referring to each item's location in the symbol table. `NAME` is the name of each symbol as it appears in the symbol table. A blank entry in this column refers to an anonymous symbol. `SCOPE` is the index of the symbol which defines the enclosing scope for a particular item. `CLASS` specifies the category of each item, e.g., type, variable, function. The `ATTRIBUTES` for each item list compiler–generated symbol information that may be useful in debugging.

Symbols in the listing are grouped according to the module, subroutine, or structure definition in which they occur. The symbol defining the module, subroutine or structure is listed first, followed by the remaining symbols in that particular scope. Symbols are numbered sequentially for easy reference to other symbols. The entry for each item includes that symbol's class and attributes, and the index of the symbol that defines that symbol's scope.

The scope of a type is either global or limited to a structure. The scope of a variable is either global (accessible anywhere), local to a compilation unit (accessible anywhere in the compilation unit which declares it), or local to a subroutine (accessible only in the subroutine that declares it).

The first group of named symbol entries in the listing for each compilation unit represents type definitions for standard built–in C types, without regard for any `-L` switch given to the compiler. Each built–in type definition entry is followed by an anonymous entry defining a pointer to that type. The built–in and pointer type definitions are followed by entries for the symbols defined in the compilation unit.

The second part of the symlist listing begins with the header line:

```
ALPHABETIC SYMBOL INDEX
```

This contains an alphabetized list of all symbols in the linked object module, with symbol indexes from the first part of the listing. If a symbol name is used many times (in different scopes), a list of indexes is given. Symbols without names are listed first under the name (anonymous).

Example

Consider the following program, `sym.c`:

```
struct structtype {
   int structint;
   char structchar;
}  mystruct;
int i;

subr()
{
   int loci;
   i = loci = 1;
}
```

Suppose we compile `sym.c` with the `-d` switch, link, and locate. The listing generated by `symlist` for the located object module is shown below:

```
INDEX NAME                SCOPE CLASS, ATTRIBUTES

    1 sym.c                   0 module, source file, line# and addr of code
stmts:
"sym.c"
       7 #00000006   10 #00000006   11 #0000000c

    2 unsigned char/short     1 type, size=1, unsigned 8 bit
    3                         1 type, ptr to type=2, size=4
    4 signed char/short       1 type, size=1, signed 8 bit
    5                         1 type, ptr to type=4, size=4
    6 int                     1 type, size=2, signed 16 bit
    7                         1 type, ptr to type=6, size=4
    8 unsigned int            1 type, size=2, unsigned 16 bit
    9                         1 type, ptr to type=8, size=4
   10 long                    1 type, size=4, signed 32 bit
   11                         1 type, ptr to type=10, size=4
   12 unsigned long           1 type, size=4, unsigned 32 bit
   13                         1 type, ptr to type=12, size=4
   14 float                   1 32 bit floating point
   15                         1 type, ptr to type=14, size=4
   16 double                  1 64 bit floating point
   17                         1 type, ptr to type=16, size=4
   18 structtype              1 type, record, size=4
   19 structint              18 field, type=6, offset=0
   20 structchar             18 field, type=2, offset=2
   21 mystruct                1 variable, type=18, static, addr=#00000000
   22 i                       1 variable, type=6, static, addr=#00000004
   23 subr                    1 function, return type=6, #args=0,
addr=#00000006
   24 loci                   23 variable, type=6, local, register=D1

    ALPHABETIC SYMBOL INDEX

i                        22
loci                     24
mystruct                 21
structchar               20
structint                19
structtype               18
subr                     23
sym.c                     1
```

6.3.5 Messages

A list of error messages for the symbol list utility appears below.

Warnings

warning— redefinition of <>

Errors

unable to open input file *file*

syntax error: token *token*, line *line*

6.4 Object Size List Utility

Function: Display size information from object module(s).

Invocation:

`olsize` *obj1* [*obj2* ...] [*options*]

Input: Standard input or *obj1* [*obj2* ...]

Output: Standard output

6.4.1 Object Size List Utility Options: Summary

The object size list utility recognizes the following options:

Option	Function	See Page:
`-0`	(Zero) Display the version number of the executable.	6–30
`-i` [*ifn*]	Take the names of input object modules from file *ifn*. If *ifn* is omitted, read the names of object modules from standard input.	6–31
`-o` *ofn*	Write output to file *ofn*.	6–31

6.4.2 Usage

The object module size list utility, `olsize`, produces a listing of the total size of code, data, and constant data contained in a collection of object modules. The input may be any combination of unlinked object modules, linked object modules, or absolute object modules, although the program runs somewhat slower on unlinked object modules.

The type of each segment, code, constant, or data, is determined by the class name. Segments with class name "code" or "CODE" are assumed to contain code; segments with class name "constant" or "CONSTANT" are assumed to contain constant data. All other segments are judged to contain data.

Example

List size of object files:

```
olsize prog1.ln prog2.ln prog3.ln
```

- Read object modules `prog1.ln`, `prog2.ln`, and `prog3.ln`.
- Write listing to the terminal.

6.4.3 Object Size List Utility Options: Detailed Descriptions

This section describes the object size list utility options in more detail and provides examples of their use.

`-0` (Zero) Display the executable's version number (for technical support use).

`-i` [*ifn*] This switch specifies that the names of input object modules are to be taken from file *ifn*. The input module names should be listed in the file, one per line. If no file is given as an argument to the switch, the names of the files are read from standard input.

`-o` *ofn* This switch specifies the name of the output file.

Example

Produce object size listing in a file:

```
olsize sample.ab hello.ol -o size.out
```

- Read object modules `sample.ab` and `hello.ol`.
- Write listing to file `size.out`.

File `size.out` is listed below:

```
Code  Data  Constant Total Hex   File
2500  2500  0        5000  1388  sample.ab
28    0     16       44    2c    hello.ol
2528  2500  16       5044  13b4  Grand Total
```

6.4.4 Messages

Severe Errors

olsize: can't read file *file*

Fatal Errors

olsize: can't read *file*, the file containing file names; aborting

olsize: can't write to *file*, the output file; aborting

CHAPTER 7

Application Notes

This chapter contains application notes on the following topics:

- Downloading
- Linking C and Assembly
- Pragma Separate (Option Separate)
- Building Libraries That Do Not Use A5
- Position–independent Code
- Getting the Best Code for Your Application
- Support for the On–board Peripherals of the 68332, 68340, and 68360
- InterTools FLEXlm Guide (Unix only)

7.1 About the Application Notes

The following is a brief summary of the contents of the application notes chapter. After this list, the remainder of the chapter is dedicated to a more detailed discussion of each topic. The following topics are covered in this chapter:

Downloading

Downloading is the process by which a program developed with InterTools is loaded into memory for the target microprocessor. The program may be downloaded to an emulator for integration and testing of the program, or directly to the user's actual hardware system. Downloading can be accomplished in a variety of ways; several methods are explained in the *Downloading* application note.

Linking C and Assembly

Interfacing C and Assembly allows the user to utilize the benefits of both languages in programming tasks. The *Linking C and Assembly* application note gives linking methods, conventions, and examples.

Pragma Separate (Option Separate)

The `#pragma separate` compiler directive is one of the InterTools C language extensions. This feature gives you complete control over the placement of global data. The *Pragma Separate (Option Separate)* application note gives a detailed explanation of this useful directive.

Building Libraries That Do Not Use A5

By default the compiler uses A5–relative addressing to access non–separate global and static variables. Some users may wish to use direct addressing instead. This can be accomplished for user code with command line switches like `-sd`. However, A5–relative references would still remain because of the compiler run–time library. This application note tells you how to build a library that does not use any A5–relative addressing.

Position–independent Code

To say that a unit of code or data is "position–independent" means that it can be moved from one location in memory to another without relinking and still execute properly. This application note describes how position independence is achieved in general, in what circumstances it would be used, and the nature of the compiler support for position independence.

Getting the Best Code for Your Application

The InterTools compiler has many options which affect optimization. Most of these options disable optimization in order to make the code easier to debug. However, there are circumstances where tradeoffs must be made. In these cases the right choice depends on your particular application. This application note describes the issues involved to help you make the best choices.

Support for the On–board Peripherals of the 68332, 68340, and 68360

The run–time libraries contain many files to support access, via C or assembly language, to the on–board peripheral units of the 68332, 68340, and 68360 processors. This application note describes how access to the peripheral components is achieved through the use of C and assembly language include files.

InterTools FLEXlm Guide (Unix only)

This application note discusses Highland Software's Flexible License Manager and how it is integrated into the InterTools Toolkit. It also contains descriptions of the Flexible License Manager license administration tools that are included with the Toolkit, runtime error messages that may occur during operation, the daemon log file and its contents, providing for uninterrupted access to the InterTools Toolkit, and the use of daemon options files to customize your use of the InterTools Toolkit.

7.2 Downloading

7.2.1 Introduction

This section discusses some of the different environments available for loading and executing programs developed with InterTools. This discussion deals with simulators, emulators, and PROM programmers.

Simulators

Simulators are software products which run on the host computer. Simulators can "simulate" program execution by converting the instructions generated for the target microprocessor into one or more instructions for the host computer. Target memory and registers are also simulated on the host. The simulator's debugger can display and modify this simulated target. Breakpoints can be set and single stepping can be done.

This technique provides a reasonable method for algorithm analysis. However, it is limited in some respects. Real time control is difficult and hardware timing tests cannot be done. For many applications an expensive test bed must be prepared to handle input and output requirements. Memory mapping is not easily done.

Emulators

Emulators are hardware devices connected to the circuit being tested. Emulators take the place of the actual target chip and provide the most thorough testing environment, since the testing is done on the actual target board. Furthermore, an emulator provides on–board firmware which can display the state of the executing program, set breakpoints and stop and start execution. Depending upon the emulator, these debugging aids can be quite elaborate.

The InterTools source–level debugger, CrossView Pro, can intelligently control an emulator during testing. The debugger provides a high–level interface between the user working on the host system's keyboard and the emulator. The user may issue commands which refer directly to the variables, source files, and line numbers as they appear in the source program. Using symbol information retained during compilation, the debugger translates the high level commands into a series of low level commands that are understood by the emulator.

Transferring a program image developed under InterTools into an emulator is a straightforward procedure. The emulator User's Manual will describe in detail how to do it. Generally the process is as follows:

The emulator is connected to the host with an RS–232 connection, usually from a terminal port on the host to a "computer" or "host" port on the emulator.

When the emulator and the host are suitably connected, the communications software can transfer the hex file produced by the formatter through the emulator into target memory.

Note: Depending upon the configuration, either the host or the emulator must have a communications package capable of monitoring and controlling the transfer of information. CrossView Pro contains one such communication package; many emulator manufacturers provide their own package.

7.2.2 PROM Programming

Once a system is thoroughly tested using an emulator, production of the finished product can be done. In embedded systems this frequently involves programming "read–only memory" chips (ROMs). This "burning" of the memory chip is done on a PROM burner.

Most PROM burners accept one or more industry standard formats. The formatter is capable of generating many standard formats. The PROM burner is connected to the host system in much the same way as the emulators described above.

There are two possible complications in the loading of the ROM memory.

One complication is encountered when the target microprocessor's data bus is wider than 8 bits, and the PROMs to be used are 8 bits wide. If the data bus is 16 bits wide, then two PROMs may be addressed in parallel. If the data bus is 32 bits, then four PROMs may be addressed at once. In this case it is necessary to distribute the bytes alternately among all the PROMs. This is known as byte–slicing. The formatter will create individual files for each PROM. For example, if 8–bit PROMs are to be programmed for a 16–bit data bus, the formatter would be run twice:

```
form file.ab -b 0 2 -o file.even
form file.ab -b 1 2 -o file.odd
```

If 8–bit PROMs are to be programmed for a 32–bit bus, the formatter would be run four times:

```
form file.ab -b 0 4 -o file.zero
form file.ab -b 1 4 -o file.ones
form file.ab -b 2 4 -o file.twos
form file.ab -b 3 4 -o file.threes
```

Each of the files would be programmed into a different PROM. Each PROM would be plugged into the appropriate socket on the target board, resulting in a complete hex image at the target level.

The second complication is also related to the need to fit a hex image into multiple PROMs. Suppose a chip has a one megabyte address space, to be filled with 256K–byte PROMs. A 256K–byte PROM has 18 address lines. The two high–order bits of the 1M–byte address are decoded by other hardware on the target board. Internally each PROM is addressed 0 to 0x3FFFF. It is necessary to break up the hex image into four different files to be burned into PROMs. The formatter can separate the full program into files of the appropriate size and can also generate the correct offset, so that the address 0x40000 will correspond to 0 when the PROM is burned.

The following commands would result in four hex files, each appropriate to burn into a PROM which is addressed on a 256K–byte boundary:

```
form file.ab -w 40000 -o first.hex
form file.ab -w 40000 -a 40000 -o second.hex
form file.ab -w 40000 -a 80000 -o third.hex
form file.ab -w 40000 -a C0000 -o fourth.hex
```

InterTools also provides the tools needed to store initialized data values in ROM, and, when the system is turned on, to automatically transfer these initial data values to RAM. The *Linking Locator* chapter describes this process (ROM processing) in greater detail.

7.3 Linking C and Assembly

7.3.1 Introduction

Interfacing C and assembly code is an important aspect of efficient programming. Modern programming experience indicates that programs written in higher level languages are more portable and reliable. Nevertheless, assembly language still offers the maximum in efficiency and flexibility. Furthermore, some machine–dependent operations cannot be performed at all in C. The combination of the two languages gives the programmer great control over execution of the task at hand.

The information in the *Compiler Run Time Conventions* and *Compiler Naming Conventions* appendices are critical to interfacing C routines and assembly language. Please refer to these sections for more information on each subject.

7.3.2 Conventions

In accordance with the compiler naming conventions, an underscore ('_') must be prepended to each procedure name in the assembly module(s) which contain(s) global symbols so that C programs and assembly modules can be linked.

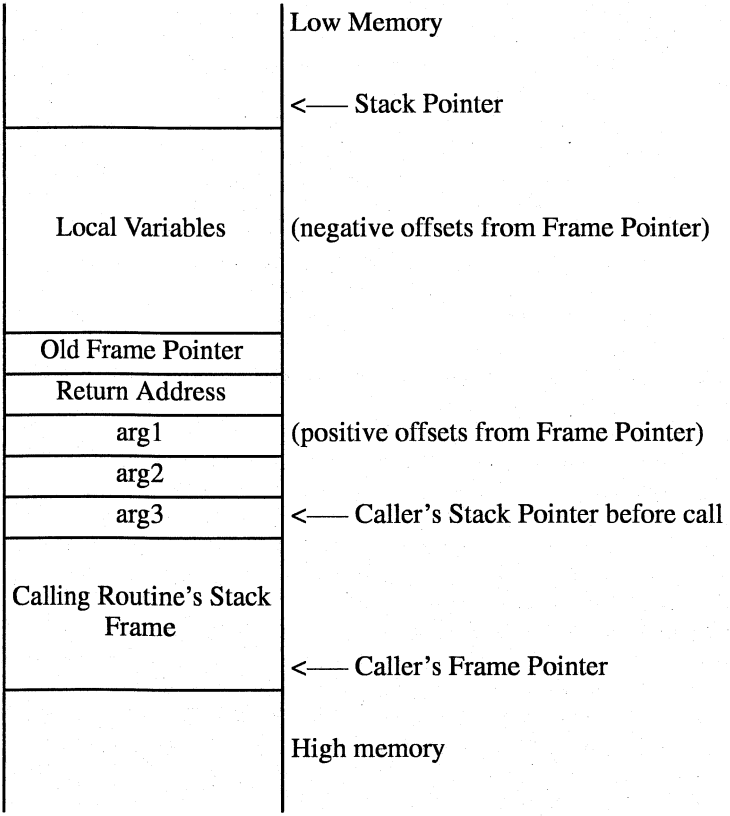
Example

Here is a sample C program which calls an assembly language routine:

```
extern void asmsub();       /* NO underscore here!
*/
main()
{
        .
        .
   asmsub(arg1,arg2,arg3); /* NO underscore here!
*/
        .
        .
}
```

After the entry code (or prologue) in the assembly routine has been executed, we obtain the following stack configuration:

Example of Stack Management:



Note: The "Calling Routine" is the C routine. The stack grows from high to low memory.

The assembly language routine `asmsub` needs to have the following form to be callable from C: (The prologue and epilogue are shown in the *Compiler Run Time Conventions* appendix.)

```
    XDEF _asmsub                /* Note the underscore
*/
_asmsub:
    [PROLOGUE]
        .
        .                       /* Body of asmsub */
        .
    [EPILOGUE]
```

Both the caller's frame pointer and the return address are 32 bits wide.

Here is an example of how to reference parameters from the stack. For example, suppose the first parameter is a pointer variable. The following instruction would be used to load the first parameter into a register:

```
MOVEA.L 8(A6),D0
```

The long word move is used since the parameter is a 32–bit integer. The source operand is 8(A6) which signifies that the first parameter is 8 bytes away from the frame pointer, A6. The result is moved into register D0.

For an example of returning a value from a function call, consider the following:

```
extern int asmfunc();
{
main()
{
   int i;
   i = asmfunc();
       .
       .
       .
}
```

When returning an integer as in the example above, the assembly routine `asmfunc` must place the return value in register D0 since the calling C routine expects the value in D0.

Notes

- Both compilations and assemblies create object modules. The linking locator takes either kind of object module as input. This provides the flexibility for linking C and assembly.
- Do not change the value of register A5! If, for example, an interrupt gives control to the C program, the C program expects register A5 to be pointing at the global data area at all times. Changing the A5 register may result in access to unexpected areas of memory.
- Pay special attention to stack management when interfacing C and assembler code. It is good practice to use macros to provide standard prologue and epilogue sequences in all assembler routines. If the hardware stack is not handled properly, the calling program may not execute correctly.
- Other conventions apply to routines returning struct types. See the *Compiler Run Time Conventions* appendix for more details.

7.3.3 Sharing Global Data

It is possible to declare data in C and reference it in assembly language and vice versa. Generally it is preferable to declare the data in C. Here are some examples showing how this is done.

Referencing C Data in Assembly Code

Consider the following C declarations:

```
long global_var;
#pragma separate sep_var
long sep_var;
```

Here we have two variables, `global_var` and `sep_var`, both of which represent 32 bit integers. According to the compiler naming conventions, these variables give rise to global symbols whose names are `_global_var` and `_sep_var` respectively. The variable `global_var` will be allocated in the `udata` segment and the variable `sep_var` will be allocated in the `s_sep_var` segment. (If `global_var` had been initialized, it would have been allocated in the `idata` segment).

Here is sample assembly language code which references these variables. It stores the value "1" in `global_var` and the value "2" in `sep_var`.

```
XREF   _global_var
XREF   _sep_var
XREF   data
   .
   .
   .
MOVE.L    #1,_global_var-data(A5)
MOVE.L    #2,_sep_var
```

Note here that the non–separate variable `global_var` is accessed through the A5 register. This register contains the address of the data group, which contains the `udata` segment containing `global_var`. Since the `_global_var` global symbol represents the full 32 bit address of the variable `global_var`, the origin of the data group must be subtracted to yield the offset relative to this register.

The non–separate variable `sep_var` is not accessed through the A5 register, because `sep_var` does not lie in the data group.

Referencing Asm Data from C

Normally it is preferable to declare the data in C, but it is still possible to reference assembly language data from C if it is properly declared.

The procedures are similar to those discussed above, except that external declarations are replaced by definitions and vice versa. The C program would look like this:

```
extern long global_var;
#pragma separate sep_var
extern long sep_var;
    .
    .
    .
global_var = 1;
sep_var = 2;
```

The assembly language declarations would look like this:

```
        XDEF      _global_var
        XDEF      _sep_var
        SECTION   udata,,'data'
_global_var DC.L   0
        SECTION   S_sep_var,,'usep'
_sep_var    DC.L   0
```

Note that the non–separate variable `global_var` is allocated in the `udata` segment. If `global_var` had an initial value, it would be allocated in `idata`. The separate variable `sep_var` is allocated in its own segment.

The name of the segment for `sep_var` chosen here matches the compiler convention, but this is not strictly necessary for the compiled code to access the variable successfully.

7.4 Pragma Separate (Option Separate)

7.4.1 Introduction

The InterTools C compiler supports a directive, `#pragma separate`, as part of the C language extensions. This directive allows the user complete control over the segmentation of global data. The older syntax `#option separate` is equivalent to `#pragma separate`.

The InterTools compiler creates a "data" group containing all data which is accessed by A5 register–relative addressing. This is a very efficient addressing mode. By default, global and static data items are allocated in one of two data segments: "`idata`," for initialized data, or "`udata`," for uninitialized data. Data allocated in these segments is restricted to a **total** size of 64K bytes because the register–relative addressing uses a 16–bit offset.

To allow for global data items that are very large, or items that the user would like to place in specific memory locations, InterTools provides the concept of separate data. **Separate data items** are placed in their own segments, and thus may be placed independently in memory using the locate function of the linking locator. Furthermore, they are not subject to the 64K–byte total size restriction imposed on normal global data. There is no limit to the size of an individual separate data item, other than the size of the memory area in which the data is allocated..

Note: The compiler uses different code sequences to access variables in the idata/udata area than it uses to access separate variables. For this reason the compiler must know whether an external variable is defined as **separate**. It is a good idea to keep the necessary `#pragma separate` directives together with "extern" declarations for separate variables.

The `#pragma separate` directive can also be used to separate constant data from read–write data. This allows constant data to be placed in ROM.

7.4.2 Preprocessor Option Directives

```
#pragma separate variable_name [segment segname]
   [class classname]
```

```
#pragma sep_on [segment segname [segname2]]
   [class classname [classname2]]
```

```
#pragma sep_off
```

The `#pragma separate` directive causes the data item named *variable_name* to be **separate**. The `#pragma separate` directive for *variable_name* must precede the definition of *variable_name* in the source program. The compiler will put this data in the specified class and segment, if that information is included in the statement.

The `#pragma sep_on` and `#pragma sep_off` directives automatically cause all global or local static data declared between them to be **separate**. This provides a shorthand notation that is equivalent to writing several `#pragma separate` directives. Note that this includes data declared with the `const` type qualifier.

The optional `segment` and `class` specifiers permit the user to specify the segment name and/or the class of the segment in which the **separate** variable is allocated. More than one **separate** variable may be allocated into a single segment, but all `#pragma separate` directives with the same `segment` option must have the same `class` option. Of course, if two **separate** variables lie in the same segment then they cannot be placed in memory independently of one another.

When only *segname* is supplied with the `segment` option, data will be allocated into *segname*. When both *segname* and *segname2* are supplied, initialized data will be allocated into *segname*, and uninitialized data will be allocated into *segname2*. If the keyword `default` replaces *segname* or *segname2*, data will be allocated into the default segments for **separate** data, as explained in the *Compiler Naming Conventions* appendix.

When only *classname* is supplied with the class option, separate segments will be given class {*classname*}. When both *classname* and *classname2* are supplied, initialized data segments will be given class {*classname*}, and uninitialized data segments will be given class {*classname2*}. If the keyword `default` replaces *classname* or *classname2*, separate segments will be given the default class name for **separate** data.

If no segment or class name is supplied, the compiler will use default names which are described in detail in the *Compiler Naming Conventions* appendix. The compiler's default rules will give rise to a different segment name for each separate variable.

7.4.3 Command Line Switches

Data can also be made **separate** by using command line switches:

`-cs` Place all data declared as `const` into a **separate** segment *cdata* and class *constant*. The `-sc` and `-ss` switches override the `-cs` switch and make its use invalid. Refer to the *C Language Specifications* appendix for an explanation of the `const` type qualifier.

`-sc` *myclass* [*myclass2*]
If the only argument is *myclass*, separate data segments will be given class {*myclass*}. If arguments *myclass* and *myclass2* are used, initialized data segments will be given class {*myclass*}, and the uninitialized data segments will be given class {*myclass2*}.

`-sd` Make all global and static data **separate**.

`-ss` *mysegment* [*mysegment2*]
If the only argument is *mysegment*, data will be allocated into *mysegment*. If arguments *mysegment* and *mysegment2* are used, initialized data will be allocated into *mysegment*, and uninitialized data will be allocated into *mysegment2*.

The command line options are equivalent to inserting the appropriate `#pragma sep_on` directive described above before the first line of the source file being compiled.

The segments defined for **separate** data items can be located either by their segment names, or by the segment class. Please see the *Linking Locator* chapter for details on how to place segments in memory.

7.5 Building Libraries That Do Not Use A5

Sometimes users may wish to eliminate the A5–relative addressing that is used by default for global data. This can be accomplished with command line switches which have the effect of making all global data "separate", that is, directly addressed. However, A5–relative addressing may still remain, because the runtime library has some private global data of its own which is addressed via A5. This application note tells you how you can build a library that does not use any A5–relative addressing.

The runtime library contains both C and assembly source files. The C modules must be recompiled using switches that make all data separate, plus any other switches, such as `-L`, that are needed for your particular application. Subsequent links will run faster if you pre–link each object module by itself (to resolve internal references), but this is not essential. Alternate versions of the assembly language modules that reference data via A5 are supplied with the library distribution, so no assemblies need be done. After the C modules are recompiled, the library index files must be updated to include the alternate assembly language modules and the recompiled C modules.

There are several different libraries supplied with the distribution. There are libraries for different targets, like MC68000 or MC68020. There may be libraries for hardware and software floating point. Within each of these are libraries for use with `-L` (long integer) and libraries for use without `-L` (the default). There are libraries that do not support floating point (the "no–floats" libraries), and libraries that do support floating point (the default). Of course, you need only rebuild the libraries you intend to use. The examples that follow show you how to rebuild all the libraries for the MC68000 software floating point target. The procedure for other targets is similar, except that you use a different compiler name, for the specific target (e.g., 68020, 68040, etc.). In addition, the command line switch for hardware floating point (`-h`) must be used for libraries which rely on floating point hardware.

At the end of this chapter there are five tables labelled "Table 1" through "Table 5". Each table represents a subset of target chips. Find the table that contains your target microprocessor. Within the table, in the 1st column find the library index file you plan to use. In the 2nd column you will find the library modules which must be replaced.

The following steps will update existing library index files. If you want to save the original index files, you should first copy the runtime library directory, then set your working directory to that directory. Otherwise, set your working directory to the runtime library. We will start with the 68000 libraries (`lib000`, `lib000.nf`, `lib000.l`, `lib000.lnf`).

1. Recompile all the C source files in the library, using `-ss libdata -sd`. This makes all library data separate in segment `libdata`. Execute a command like this for each c file:

```
c68000 abort.c -ss libdata -sd
```

2. Pre–link each object module. Execute a command like this for each `.ol` file you just created:

```
llink strcpy.ol -lo -w -o strcpy.ln
```

3. Update the library index file. Table 1 contains the library index files for target MC680000. Looking up `lib000` in the table we find the following modules must be replaced: `adln.ln`. `adlog.ln`, `adsqrt.ln`, `dpfncs.ln`, `dpopns.ln`, `fpopns.ln`, `pmain.ln`, and `xlfncs.ln`. Execute the following commands:

For the PC:

```
libr -L lib000
   -d adln.ln adlog.ln adsqrt.ln dpfncs.ln
   dpopns.ln fpopns.ln pmain.ln xlfncs.ln
   -a adlnx.ln adlogx.ln adsqrtx.ln dpfncsx.ln
   dpopnsx.ln fpopnsx.ln pmainx.ln xlfncsx.ln

libr -L lib000 -u
```

For Unix hosts:

```
libr -L lib000 \
   -d adln.ln adlog.ln adsqrt.ln dpfncs.ln \
   dpopns.ln fpopns.ln pmain.ln xlfncs.ln
   -a adlnx.ln adlogx.ln adsqrtx.ln dpfncsx.ln
\
   dpopnsx.ln fpopnsx.ln pmainx.ln xlfncsx.ln

libr -L lib000 -u
```

This replaces the assembly language modules with the "no–A5" versions. All the "no–A5" versions have names that end in "x". It also updates the library with the recompiled versions of the C modules.

Now we continue with the "no–floats" library.

4. Recompile `xprintf` and `xscanf`. Here you need the extra command line option `-P NO_FP_IO` to generate the "no–floats" version of `xprintf` and `xscanf`:

```
c68000 xprintf.c xscanf.c -P NO_FP_IO -ss libdata -sd
```

5. Pre–link `xscanf` and `xprintf`. Execute the following commands. Note the output going to the special `.in` suffix:

```
llink xprintf.ol -lo -w -o xprintf.in
llink xscanf.ol -lo -w -o xscanf.in
```

6. Update the no–floats library index file. This is similar to step 3), except that the only "no–A5" assembler module in the no–floats library is `pmain`:

```
libr -L lib000.nf -d pmain.ln -a pmainx.ln
libr -L lib000.nf -u
```

Now we continue with the long integer libraries. Here the procedure is similar, except in the following ways. An additional switch, `-L` is required on all compilations. The list of "no–A5" assembler modules is different. The library index file names are different. The linked module suffix names `.ln` and `.in` are replaced with `.lln` and `.iln`.

7. Recompile all the C source files in the library, using `-sd, -ss libdata` and `-L`. Execute a command like this for each c file:

```
c68000 abort.c -L -ss libdata -sd
```

8. Pre–link each object module same as step 2), except direct the output to `.lln`, not `.ln`, for example:

```
llink strcpy.ol -lo -w -o strcpy.lln
```

9. Rebuild the library index files. Again refer to Table 1 to find which modules must be replaced in library index file `lib000.l`. Execute these commands:

For the PC:

```
libr -L lib000.l -d adlnl.ln adlogl.ln
   adsqrtl.ln dpfncs.ln dpopns.ln
   fpopns.ln pmain.ln xlfncs.ln
   -a adlnlx.ln adloglx.ln
   adsqrtlx.ln dpfncsx.ln dpopnsx.ln
   fpopnsx.ln pmainx.ln xlfncsx.ln

libr -L lib000.l -u
```

For Unix hosts:

```
libr -L lib000.l -d adlnl.ln adlogl.ln \
   adsqrtl.ln dpfncs.ln dpopns.ln \
   fpopns.ln pmain.ln xlfncs.ln
   -a adlnlx.ln adloglx.ln \
   adsqrtlx.ln dpfncsx.ln dpopnsx.ln \
   fpopnsx.ln pmainx.ln xlfncsx.ln

libr -L lib000.l -u
```

This replaces the assembly language modules with the "no–A5" versions. It also updates the library with the recompiled versions of the C modules. Now we continue with the long integer, no–floats library. Note that the list of assembly language modules is different here than it was in the short integer library. For example, `lib000.l` uses `adsqrtl.ln` whereas `lib000` uses `adsqrt.ln`. Again, refer to Table 1 to find which modules must be replaced in library index file `lib000.l`.

10. Recompile no–floats, long integer version of `xprintf` and `xscanf`:

```
c68000 xprintf.c xscanf.c -P NO_FP_IO -ss libdata -L -sd
```

11. Pre–link `xscanf` and `xprintf`. This is the same as step 5), except direct the output to `.iln` instead of `.in`, that is:

```
llink xprintf.ol -lo -w -o xprintf.iln
llink xscanf.ol -lo -w -o xscanf.iln
```

12. Update the no–floats library index file:

```
libr -L lib000.lnf -d pmain.ln -a pmainx.ln
libr -L lib000.lnf -u
```

If you are using a C++ library (`cpp000.lib` or `cpp020.lib`), it will need to be updated as well. Remember that for MC68000, MC68EC000, MC68008, MC68010, and MC68302 targets the `cpp000.lib` library is used; otherwise the `cpp020.lib` library is used. Also note that since C++ assumes `-L`, a C long library must also be created as shown above. This example includes a rebuild of `cpp000.lib`.

1. Recompile all C++ source files using `-sd` and `-ss` libdata. Execute a command like this for each C++ file:

```
cp68000 array_del.cpp -sd -ss libdata -I . ..\cppinc
--exceptions --building_runtime
```

The `cpp020.lib` library was oringinally built using the MC68332 compiler, but other (non–68000) compilers could be used for this purpose.

2. Recompile the C file `whatami.c`:

```
c68000 whatami.c -sd -ss libdata
```

3. Pre–link each object module. Execute a command like this for each `.ol` file for just created:

```
llink array_del.ol -lo -w -o array_del.0ln
```

The `.0ln` extension is used for `cpp000.lib` and the .2ln extension is used for `cpp020.lib`.

4. Update the library index file:

```
libr -L cpp000.lib -u
```

The procedure is the same for every other target–specific library, except that the library index files are different and the list of assembler modules containing A5 references is different.

The following table summarizes all the assembler modules that use A5–relative addressing and what library index files they belong to. In all cases the name of the replacement no–A5 version is formed by adding an "x" before the suffix. For example, the no–A5 replacement for `adsqrt.ln` is `adsqrtx.ln`.

Here is how you use this table. Look up the library you are are rebuilding on the left–hand column. Update the library index file by deleting the named modules and adding their replacements. There is one table for each library directory which covers all the library index files in that directory.

Table 1 (MC68000, MC68010, MC68302)				
Libraries:		**Modules to be Replaced with no–A5 Versions:**		
lib000	lib010	adln.ln	adlog.ln	adsqrt.ln
lib302	lib020s	dpfncs.ln	dpopns.ln	fpopns.ln
lib030s		pmain.ln	xlfncs.ln	
lib000.nf	lib000.lnf	pmain.ln		
lib010.nf	lib010.lnf			
lib302.nf	lib302.lnf			
lib020s.nf	lib030s.lnf			
lib000.l	lib010.l	adlnl.ln	adlogl.ln	adsqrtl.ln
lib302.l	lib020s.l	dpfncs.ln	dpopns.ln	fpopns.ln
lib030s.l		pmain.ln	xlfncs.ln	
lib302ap		adln.ln	adlog.ln	adsqrt.ln
lib302at		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn302a.ln	xlfncs.ln	
lib302ap.nf	lib302ap.lnf	pmn302a.ln		
lib302at.nf	lib302at.lnf			
lib302ap.l		adlnl.ln	adlogl.ln	adsqrtl.ln
lib302at.l		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn302a.ln	xlfncs.ln	
lib000r		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmainr.ln	xlfncs.ln	
lib000r.nf		pmainr.ln		

Table 1 (MC68000, MC68010, MC68302) — (*Continued*)

Libraries:	Modules to be Replaced with no–A5 Versions:		
lib000r.l	adlnl.ln	adlogl.ln	adsqrtl.ln
	dpfncs.ln	dpopns.ln	fpopns.ln
	pmainr.ln	xlfncs.ln	

Table 2 (MC68020, MC68030, MC68332, MC68340, MC68360), no 68881/68882				
Libraries:		**Modules to be Replaced with no–A5 Versions:**		
lib020s	lib030s	adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmain.ln	xlfncs.ln	
lib020s.nf	lib020s.lnf	pmain.ln		
lib030s.nf	lib030s.lnf			
lib020s.1	lib030s.1	adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmain.ln	xlfncs.ln	
lib332		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn332.ln	xlfncs.ln	
lib332.nf	lib332.lnf	pmn332.ln		
lib332.1		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn332.ln	xlfncs.ln	
lib340		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn340.ln	xlfncs.ln	
lib340.nf	lib340.lnf	pmn340.ln		
lib340.1		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn340.ln	xlfncs.ln	

Table 2 (MC68020, MC68030, MC68332, MC68340, MC68360), no 68881/68882 — (*Continued*)

Libraries:		Modules to be Replaced with no–A5 Versions:		
lib340b		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn340b.ln	xlfncs.ln	
lib340b.nf	lib340b.lnf	pmn340b.ln		
lib340b.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn340b.ln	xlfncs.ln	
lib360		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn360.ln	xlfncs.ln	
lib360.nf	lib360.lnf	pmn360.ln		
lib360.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn360.ln	xlfncs.ln	
lib360b		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn360b.ln	xlfncs.ln	
lib360b.nf	lib360b.lnf	pmn360b.ln		
lib360b.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn360b.ln	xlfncs.ln	

Table 2 (MC68020, MC68030, MC68332, MC68340, MC68360), no 68881/68882 — (*Continued*)				
Libraries:		**Modules to be Replaced with no–A5 Versions:**		
lib332r		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn332r.ln	xlfncs.ln	
lib332r.nf	lib332r.lnf	pmn332r.ln		
lib332r.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn332r.ln	xlfncs.ln	
lib030r		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn030r.ln	xlfncs.ln	
lib030r.nf	lib030r.lnf	pmn030r.ln		
lib030r.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn030r.ln	xlfncs.ln	

Table 3 (MC68020, MC68030), with 68881/68882				
Libraries:		**Modules to be Replaced with no–A5 Versions:**		
lib020h	lib030h	acos.ln	asin.ln	log.ln
		log2.ln	log10.ln	pmain.ln
		sqrt.ln		
lib020h.nf	lib020h.lnf	pmain.ln		
lib030h.nf	lib030h.lnf			
lib020h.l	lib030h.l	acosl.ln	asinl.ln	logl.ln
		log2l.ln	log10l.ln	pmain.ln
		sqrtl.ln		
lib030hr		acos.ln	asin.ln	log.ln
		log2.ln	log10.ln	pmn030r.ln
		sqrt.ln		
lib030hr.nf	lib020hr.lnf	pmn030r.ln		
lib030hr.l		acosl.ln	asinl.ln	logl.ln
		log2l.ln	log10l.ln	pmn030r.ln
		sqrtl.ln		

Table 4 (MC68EC040)				
Libraries:		**Modules to be Replaced with no–A5 Versions:**		
libe40		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmain.ln	xlfncs.ln	
libe40.nf	libe40.lnf	pmain.ln		
libe40.1		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmain.ln	xlfncs.ln	

Table 5 (MC68040)				
Libraries:		**Modules to be Replaced with no–A5 Versions:**		
lib040		acos.ln	asin.ln	log.ln
		log2.ln	log10.ln	pmain.ln
		sqrt.ln		
lib040.nf	lib040.lnf	pmain.ln		
lib040.1		acosl.ln	asinl.ln	logl.ln
		log2l.ln	log10l.ln	pmain.ln
		sqrtl.ln		

Table 6 (MC68060)				
Libraries:		**Modules to be Replaced with no–A5 Versions:**		
lib060		acos.ln	asin.ln	log.ln
		log2.ln	log10.ln	pmainf.ln
		sqrt.ln		
lib060.nf	lib060.lnf	pmain.ln		
lib060.l		acosl.ln	asinl.ln	logl.ln
		log2l.ln	log10l.ln	pmainf.ln
		sqrtl.ln		

7.6 Position–independent Code

7.6.1 Introduction

The most common situation where position independence is desirable is in conjunction with an operating system that supports the concept of a dynamically loaded program. When the system is built, the dynamically loaded program is compiled and linked, and then stored on an external device. At runtime the program is loaded into memory by the system loader and executed.

If the address of the dynamically loaded program can be predicted in advance, then there is no need for position independence. One merely has to locate the program at the address where it will be when it is executed. This really is more like an overlay than a dynamically loaded program. If the execution address is not determined until run time, then the linker cannot really know the absolute addresses of the code or data segments in the program.

There are two general approaches which can be used. The first is to arrange for the loader program to know where the address references are in the program being loaded, and to have the loader update these references as the program is being loaded. In some sense the loader completes the work of the link editor, and so one says that the loader is acting as a "linking loader". Systems that use a linking loader also do not need position independence.

The alternative is to have the loader do no address correction during the loading operation. However, since there is no way to predict where the program will be loaded, there is certain to be a mismatch between the addresses where the link editor thought the program would be loaded and the address where it actually will be loaded. To say that the program is position independent is equivalent to saying that the program will execute properly even under these circumstances.

As will be explained below, position–independent code generation patterns are less efficient than the patterns used by default. Generally speaking, the cost is quite moderate if the application is small (less than 32K bytes). Larger applications, especially on a MC68000 rather than a MC68020 or CPU32 processor, may suffer a larger performance degradation. If these costs are not acceptable, then the best recourse is to implement a linking loader.

It does require some work to implement a linking loader. There is no simple industry standard download format for a relocatable program image. For absolute program images there are many standards (Motorola S–records, Tekhex, and so on). Consequently, users who wishes to use a linking loader must design their own download format, write their own loader, and write a utility program that converts the InterTools relocatable module file (the `.ln` file) into their relocatable download format. While straightforward, these tasks do require some effort. In contrast, writing a non–linking loader program that reads a standard absolute format like S–records is very simple.

7.6.2 How Position Independence is Achieved

Programs cannot always be moved from place to place and still execute correctly because references to code or data may become incorrect when the objects they refer to are moved. The M68000 family processors support three general forms of memory references:

- **Direct Addressing**
  The address of the memory location is expressed as a constant address. Usually this is a 32–bit constant, but it can be expressed as a 16–bit constant which is sign–extended to form the address.
- **PC–relative Addressing**
  The address of the memory location is expressed as a displacement which is added to the value in the PC at the time of the reference to form the address.

- **Register–based Addressing**
  This includes all other addressing modes. The address is usually expressed as a sum of registers and constants.

Some examples might make this more clear. Suppose we have a procedure named `f` which we wish to call. Here are three different ways we could call `f`:

```
JSR f          Direct Addressing
BSR f          PC–relative Addressing
JSR (A0)       Register–based Addressing
```

In the third case, `A0` is assumed to contain the address of `f`.

It is clear that the direct addressing case is never position independent. The instruction contains the absolute address of the object being referenced (`f`). If `f` is moved, then the reference becomes invalid. On the other hand, the instruction containing the JSR can itself be moved without causing problems. It is the destination of the call that matters.

PC–relative addressing will remain correct, as long as both the object being referenced (`f`), and the instruction making the reference, are moved by the same amount. In that case the difference between their addresses remains constant, and it is this difference that is contained in the instruction. However, if only one of the two are moved, or if both are moved by different amounts then the reference becomes incorrect.

Register–based addressing will remain correct, as long as the value in the A0 register is adjusted so that it points to the address to which `f` was moved. Naturally this depends on how A0 was set up. For example, suppose the instruction that set up A0 looked like this:

```
MOVEA.L  #f,A0
```

In this case the register–based addressing would behave just like the direct addressing case, i.e., not position independent, because the MOVEA instruction contains the absolute address of `f`. On the other hand, suppose A0 was set up like this:

```
LEA    f(PC),A0
```

Then register–based addressing would behave just like the PC–relative case.

Another alternative is for A0 to be set by the operating system. For example, suppose `f` is the main entry point into a dynamically loaded program. Then the system call that causes `f` to be loaded would presumably return the address of `f` to the caller. In that case the register–based addressing would be position–independent. It would reach `f` no matter where `f` was loaded.

The InterTools C compiler always uses register–based addressing for calls through "pointer–to–procedure" variables. By default, the InterTools compiler uses direct addressing when assigning the address of a named function to a pointer–to–procedure variable. When the position–independent code options, `-ps` or `-pc`, are present, the compiler uses PC–relative addressing instead. Therefore pointer–to–procedure variables work correctly as long as the procedure which assigns the pointer and the procedure being assigned are part of the same position–independent unit.

If an application needs to access code that is outside the position–independent unit, such as operating system services, then there are two ways to do so. One is to use a pointer–to–procedure variable which is deliberately initialized in a non position–independent manner. This could be done by C code compiled without the position–independent options, or by assembly language using direct addressing. The other is to use TRAP instructions and interrupt handlers.

7.6.3 Position Independence and Data References

By default, the InterTools compiler references most data via register–based addressing. By default, global and static data is referenced via A5. Separate data and string literals (quoted strings), on the other hand, are referenced via direct addressing. Since direct addressing is incompatible with position independence, the compiler must be forced to use either PC–relative addressing or A5–relative addressing instead.

First, let's discuss the consequences of PC–relative addressing. The `-pd` switch forces the compiler to use PC–relative addressing for data in cases where it would have used direct addressing. If you use PC–relative addressing for data, then there are three requirements:

- The data and the code must be relocated together so that their relative offset remains constant. This is not necessary when using A5–relative addressing, since the base address for data (in A5) is not related to the code.
- PC–relative global data, like directly addressed data, is not reentrant. A5–relative data, on the other hand, can be made reentrant by dynamically allocating the A5–relative data area. If your system generates multiple real–time tasks executing the same code, then you must avoid using non–constant separate data.
- If your hardware uses separate code and data address spaces, then you must also supply the `-id` switch. This switch informs the compiler that instruction and data storage is different. Since PC–relative addressing generates a code fetch, the compiler must avoid using PC–relative addressing on data fetches. However, it can still use an LEA with a PC–relative addressing mode to compute an address which it uses to reference data. Examples are supplied below showing how this works.

If these requirements are acceptable to you, then you can achieve position–independence for data merely by supplying the `-pd` switch in addition to the `-ps` or `-pc` switch.

If you cannot arrange for the code and data to be relocated together then you cannot use PC–relative addressing for data at all. In that case you must adopt a strategy of making all non–stack data A5–relative. To be precise, building a system in which all non–stack data is addressed via A5 can be achieved by the following steps:

1. Remove any `#pragma separate` or `#pragma sep_on` directives from your code. Avoid using the following compiler options: `-cs`, `-sd`, `-ss`, `-sc`, and `-pd`. This ensures that there will be no separate data in your system.
2. Supply the `-si` switch on all compilations. This causes string literals to be allocated in the idata segment.
3. If the total size of data is more than 64K bytes, supply the `-b5` option. This forces the compiler to use 32–bit A5–relative offsets instead of 16–bit offsets, and thus relaxes the 64K–byte limit on A5–relative data.
4. If the total size of code is less than 32K bytes, supply the `-ps` option; otherwise supply the `-pc` option.

If your system requires reentrancy, then you cannot use PC–relative addressing for non–constant data. Building a system in which all non–stack non–constant data is addressed via A5, but constant data is addressed via PC–relative addressing, can be achieved by the following steps:

1. Use `#pragma separate` or `#pragma sep_on` directives only when they affect read–only variables. You may also use `-cs`, which makes all const–qualified variables separate. Do not use `-sd`, `-ss`, or `-sc`.
2. Supply `-pd` on all compilations.
3. Supply `-b5` if the total size of A5–relative data is more than 64K bytes.
4. If the total size of code, separate data, and string literals is less than 32K bytes, supply the `-ps` option. Otherwise, supply the `-pc` option.

Note: Whatever method you choose, the compiler's runtime library must be rebuilt using these same procedures. See the Runtime Library appendix of the InterTools Reference Manual for details about how to recompile the runtime library. The runtime library sources do not have any non–constant separate data, but it does have both read–write A5–relative data and constant separate data.

The A5–only strategy is probably the most efficient approach if there is less than 64K bytes of A5–relative data. The 16–bit A5–relative addressing mode is quite efficient. However, if there is more than 64K bytes of A5–relative data, then the `-b5` switch is necessary, and this imposes a significant performance penalty. In that case it might be better to try to keep the A5–relative data area under under 64K by using some separate data and/or allocating string literals outside the A5–relative data area.

Here are some examples which should make it more clear how these various methods and options work. Consider these declarations and assignments:

```
#pragma separate s
long s;
long a;
s = a;
a = s;
```

Here are some typical code patterns which might be generated for these assignments under various combinations of options. Note that PC–relative addressing may not be used for destination operands, and that `-id` means that PC–relative addressing may not be used for source operands either. Also, the 68000 target does not support 32–bit offsets in addressing modes, so these addressing modes must be simplified before they can be used.

The actual addressing modes being used are shown, as well as the total count of bytes and cycles. Cycles counts were obtained by adding together the "cache case" for the 68020 processor. Timing would be different on other processors. The actual code sequences used by the compiler in any given program would depend on surrounding context and other compiler options.

Default options: 16 bytes, 17 cycles:

```
MOVE.L    _a-data(A5),_s          d16(An),d32          8/10
MOVE.L    _s,_a-data(A5)          d32,d16(An)          8/7
```

-pd, 68020: 22 bytes, 38 cycles:

```
LEA       (_s,PC),A0              (d32,PC),An          8/14
MOVE.L    _a-data(A5),(A0)        d16(An),(An)         4/8
MOVE.L    (_s,PC),a-data(A5)      (d32,PC),d16(An)     10/16
```

-pd, 68000: 26 bytes, 36 cycles:

```
MOVE.L   #_s-*-8,D0                i32,Dn              6/6
LEA      *+2(PC,D0.L),A0           d8(PC,Dn),An        4/6
MOVE.L   _a-data(A5),(A0)          d16(An),(An)        4/8
MOVE.L   #_s-*-8,D0                i32,Dn              6/6
MOVE.L   *+2(PC,D0.L),a-data(A5)   d8(PC,Dn),d16(An)   6/10
```

-pd, 68020, -id: 24 bytes, 33 cycles:

```
LEA      (_s,PC),A0                (d32,PC),An         8/14
MOVE.L   _a-data(A5),(A0)          d16(An),(An)        4/8
LEA      (_s,PC),A0                (d32,PC),An         8/14
MOVE.L   (A0),a-data(A5)           (An),d16(An)        4/7
```

-pd, 68000, -id: 28 bytes, 39 cycles:

```
MOVE.L   #_s-*,D0                  i32,Dn              6/6
LEA      *-6(PC,D0.L),A0           d8(PC,Dn),An        4/6
MOVE.L   _a-data(A5),(A0)          d16(An),(An)        4/8
MOVE.L   #_s-*,D0                  i32,Dn              6/6
LEA      *-6(PC,D0.L),A0           d8(PC,Dn),An        4/6
MOVE.L   (A0),a-data(A5)           (An),d16(An)        4/7
```

-pd, -ps: 14 bytes, 20 cycles:

```
LEA      _s(PC),A0                 d16(PC),An          4/4
MOVE.L   _a-data(A5),(A0)          d16(An),(An)        4/8
MOVE.L   _s(PC),a-data(A5)         d16(PC),d16(An)     6/8
```

-pd, -ps, -id: 16 bytes, 23 cycles:

```
LEA      _s(PC),A0                 d16(PC),An          4/4
MOVE.L   _a-data(A5),(A0)          d16(An),(An)        4/8
LEA      _s(PC),A0                 d16(PC),An          4/4
MOVE.L   (A0),a-data(A5)           (An),d16(An)        4/7
```

-b5, 68020: 24 bytes 34 cycles:

```
MOVE.L    (_a-data,A5),_s          (d32,An),d32        12/18
MOVE.L    _s,(_a-data,A5)          d32,(d32,An)        12/16
```

-b5, 68000: 28 bytes, 33 cycles:

```
MOVE.L    #_a-data,D0              i32,Dn              6/6
MOVE.L    (A5,D0.L),_s             d8(An,Dn),d32       8/12
MOVE.L    #_a-data,D0              i32,Dn              6/6
MOVE.L    _s,(A5,D0.L)             d32,d8(An,Dn)       8/9
```

Here are some of the things this table demonstrates:

- All the position–independent methods impose some penalty in size or speed or both.
- The A5 + 16–bit offset and PC + 16–bit offset addressing modes are very efficient. However, the A5 + 16–bit mode is more applicable. It can be used as source or destination, and only imposes a limit of 64K bytes on data size. The PC + 16–bit mode can only be used as a source, and even then only if the `-id` switch is absent, and it imposes a limit of 32K bytes on the total amount of code and data.
- All the other addressing modes are larger and slower, especially on the 68000 target.

The choice of whether to use A5–relative addressing or PC–relative addressing for data may also be influenced by issues of reentrancy and storage management. Generally speaking, if you have one program which may be reentered, then A5–relative data will be private (unshared), while PC–relative data will be public (shared). If your application require shared data, then you must support some PC–relative data. If your application requires private data, then you must support some A5–relative data. All things being equal, the best strategy is probably to have the constant data be shared, and the non–constant data be private. This avoids unnecessary duplication of constants.

If your system separates code and data into two different address spaces, then it may be difficult for you to cause data to be relocated together with code. In that case you cannot use PC–relative addressing for data, and must instead use the pure A5–relative strategy.

7.6.4 Position Independence and Data Initialization

In C, the initial value of a global or static variable must be a constant. Addresses of global or static variables, string literals, and functions are all considered to be constant. In a position–independent world, such addresses are only runtime constants, not link–time constants. In fact, in a program which dynamically allocates the A5–relative data area, addresses of A5–relative data items are also not really constant.

This creates special problems for data initializations which involve addresses. For example:

```
char *p = "abc";
char **q = &p;
```

Here "p" is supposed to be initialized with the address of the string of characters "abc", and "q" is supposed to be initialized with the address of "p". Under position–independence, the only way such initialization can be performed is with runtime code, because the addresses of "abc" and "p" are not known until the program begins execution. When the position–independent code options are present, the compiler will emit a warning for such initializations:

```
Address initialization not position-independent
(Warning only)
```

The only way such initializations could be made to work is if the loader program adjusts these initializations when the proper addresses become known. Of course, if the loader were capable of this, then position–independence would not be required at all. Assuming the loader is not capable of adjusting these initializations, the only other strategy is to avoid using these kinds of initializations.

If you have an address initialization, the best strategy is to replace the initialize address value with 0 (null pointer), and then assign the correct address value with an assignment statement which would be executed somewhere before the first reference to the variable. Your "main" routine would be a good spot to put such assignments.

7.6.5 Building a Position–independent System

There are many ways to build position–independent systems. The following example describes one way of doing it, but there are many alternatives. In this example, we discuss building a system consisting of a root portion which will be in the initial system load and two collections of subroutines which would be loaded dynamically.

To allow the dynamically loaded modules to use runtime library routine and system service routines in the root, we devised a system of special "`.h`" header files.

The idea behind these special "`.h`" headers is to distinguish calls to the root (which should NOT be position–independent) from calls to other modules in the same position–independent package (which should be position–independent). Since the root does not move with the position–independent module, we want calls to the root to always go to the same place. Note that we do not allow calls from one position–independent package to another.

We placed these special headers in a separate directory which we name with a `-S` option ahead of the standard runtime library when compiling our position–independent code. Here is an example showing what these special headers look like:

```
_CASM int printf(char *p, ...) {
      XREF _printf
      JSR _printf
}
```

These headers redefine the normal C library routines as in–line assembly language routines consisting only of a non–position–independent call. This "trick" allows us to code ordinary–looking calls to the root, but still get the code we require.

The data usage pattern for our application did not present any special problems. Each position–independent module had its own private data, and did not contain any references to the data in the root. Thus we were able to adopt a relatively simple convention for data. All the data belonging to each position–independent module will be made separate. It will be collected together in a single segment which will become part of the position–independent unit and will move with it as it is relocated. It will be accessed via PC–relative addressing.

Data in the root will be A5–relative. Since the runtime library routines are only called from the root, we do not need to rebuild the runtime library to make it position–independent.

Here are the steps we used to build this system:

1. Compile all the root modules with no special options.
2. Compile all the dynamically loaded modules for the first package. We supply the special header files via `-S`, and we make sure that there are no address initializations anywhere in this source. We add the `-sc p1data` switch to make all data separate, in data segments of class `p1data`. We also supply `-ps` and `-pd`, to make the code position–independent and the (separate) data PC–relative. Finally, we add `-cc p1code` to assign a uniform class name for all the code in these modules.
3. Compile all the dynamically loaded modules for the second package. We supply the same switches as in step 2, except that we

use `-sc p2data` and `-cc p2code` instead of `-sc p1data` and `-cc p1code`. This separates the code and data for this package from the code and data for the first package.

4. Link all modules together into a single `.ab` file. Use use locator commands to control the placement of segments in target memory. Our goal is to locate all the code and data in the each position–independent package in its own address range. It doesn't matter much what address range we use, because the program will not actually be loaded at that address. The following locator commands cause the first position–independent module to be located between addresses 100000 and 150000 (hex), and the second module to be located between addresses 200000 and 250000 (hex):

```
LOCATE ({p1code} {p1data} : #100000 TO
#150000);
LOCATE ({p2code} {p2data} : #200000 TO
#250000);
```

We locate the root code and data in the range of addresses where it will actually run. In our target system, we plan to have ROM in the address range between 0 and 20000 (hex). We make sure that only root code and constant data will be placed in this area. We plan to have RAM in the address ranges above 20000 (hex). We make sure that root data will be located in this area. Here are the locator commands that will do this:

```
LOCATE ({code} {constant} : #0 TO #20000);
LOCATE ({data} : AFTER #20000);
```

5. Once the link is finished, we format the resulting `.ab` file. We want to form three hex files: one for the root and one for each position–independent module. There are several ways to do this. One is to tell the formatter to format a particular address range. Another is to tell the formatter to include a named class of segments, excluding all others. Another is to tell the formatter to exclude a named class of segments, including all others. Here is an example showing how to extract the first position–independent module by including only the segments in class `p1code` and `p1data`:

```
form module.ab -ic p1code p1data -o p1.hex
```

Here the `-ic` switch specifies the classes of segments to include. Here is an example that extracts the root by excluding both position–independent packages:

```
form module.ab -ec p1code p1data p2code
p2data
     -o root.hex
```

Here the `-ec` switch specifies the classes of segments to exclude. Here is an example that extracts the second position–independent package by specifying the address range containing that module (100000 to 150000 hex):

```
form module.ab -a 100000 -w 50000 -o p2.hex
```

Here the `-a` switch defines a starting address, and the `-w` switch defines a size. So this command generates a download file that represents the values between hex 100000 and 150000.

7.6.6 Some Additional Hints

If our application did have sharing of data between the root and the position–independent modules, then our approach would have been slightly different. First we would not use the `-sc` switch when compiling the position–independent modules. By using `-sc`, we are telling the compiler that all data is separate. This is not the case anymore, since the compiler must be told that the shared root data is A5–relative.

We can still make the position–independent data separate, but we would have to use the `#pragma separate` or `#pragma sep_on` directives rather than command line switches to do it. Alternatively, we could make the position–independent data A5–relative. However, this would mean that this data would now be located in the root. It might be important to save valuable space in the root by having the data belonging to the position–independent module located with the position–independent module, outside the root.

7.7 Getting the Best Code for Your Application

The InterTools compiler has many options which affect optimization. Most of these options disable optimization in order to make the code easier to debug. However, there are circumstances where tradeoffs must be made. In these cases the right choice depends on your particular application. This application note describes the issues involved to help you make the best choices.

7.7.1 Code Size versus Execution Speed

The first thing to consider is whether your application should be optimized for execution time or code size. The `-ot` switch directs the compiler to use any means necessary to optimize for time, and the `-os` switch directs the compiler to optimize for size. The default behavior is to optimize for time, but in moderation. Certain very space–intensive optimization techniques, such as loop unrolling, are not enabled by default, only under `-ot`.

For most applications both code size and execution speed are important. Usually the best strategy in these cases is to identify which modules are executed most often and compile them to optimize execution time. Most modules are executed infrequently, and they can be optimized for code size. The conventional wisdom, sometimes referred to as the "90–10" rule, is that 90% of the time is spent executing 10% of the code.

7.7.2 If Statements

Suppose you have an `if` statement with an `else` clause:

```
if (expr) {
   true_clause;
} else {
   false_clause;
}
```

This generally compiles into code that looks like this:

test expr
branch on condition to false clause
true clause
branch unconditionally around false clause
false clause

From this example, you can see that you are better off putting your more frequently executed code in "false clause". This is because execution of the "false clause" requires one branch–taken, where execution of the "true clause" requires one branch–not–taken and one branch–taken. This is especially true for the MC68040, which favors branch–taken over branch–not–taken.

7.7.3 Using Integer Data

There are a number of coding techniques which can be used when dealing with integer variables. On the one hand, one can take care to declare each variable with the smallest type which can hold all the values you expect to store in that variable. Alternatively, you can declare all your integral variables as 32–bit integers. Which is best?

The strategy of using all 32–bit integers avoids generating any code to sign–extend 16–bit values to 32 bits. These operations would otherwise occur whenever 16–bit variables are combined with 32–bit variables. However, there are a number of inefficiencies in 32–bit variables.

On the 68000 family processors, 32–bit multiply and divide operations are very much slower than 16–bit multiply and divide. In fact, the 68000, 68010, and 68302 processors have no long multiply or divide instruction, so such operations are done out–of–line in a library routine. Naturally you will want to avoid such operations whenever possible.

The compiler does optimize divide and mod by powers of two and most multiplications by constants. For example, multiplication by four is performed using a left shift, not a multiply instruction.

Another important fact is that 32–bit constant operands are considerably more expensive than 16–bit constant operands. In fact, on all processors but the 68040 and 68EC040, this sequence:

```
MOVEQ.L   #100,D0
MOVE.L    D0,(A0)
```

is actually smaller and faster than this sequence:

```
MOVE.L    #100,(A0)
```

Of course, this simplification can only be performed if the value of the constant is between –128 and +127. Even so, this shows how slow the 32–bit constant operand is: you can run a whole extra instruction in less time than the difference between a constant operand and a register operand.

For these reasons, it is generally more efficient to use 16–bit variables than 32–bit variables. However, there is no further advantage to using 8–bit variables instead of 16–bit variables. The cost of any operation is the same whether it is done in 8 bits or 16 bits. In fact, the 8–bit operations usually are more expensive, because of the additional sign–extension operations which can occur.

In summary, we recommend:

- Only use byte variables in large arrays when the savings in data size make them worthwhile. Local variables should always be at least 16 bits wide.
- Try to avoid combining 16–bit integers with 32–bit integers. For example, suppose you have two counters which will be added together, and one of them must be 32 bits wide. Then it is probably better to make the other counter 32 bits wide also.
- Avoid 32–bit multiply and divide operations whenever possible.

7.7.4 Size of `int` Datatype

The InterTools compiler allows you to determine the size of the built–in `int` datatype (by default the size of `int` is 16 bits). It is of course possible to declare integers of any size, whether you use `-L` or not. For example, you can use `#define`'s like these:

```
#define INT32 long
#define UINT32 unsigned long
#define INT8 signed char
#define UINT8 unsigned char
```

If you use `-L`, you would complete the set with these `#define`'s:

```
#define INT16 short
#define UINT16 unsigned short
```

If you do not use `-L`, you would use these `#define`'s instead:

```
#define INT16 int
#define UINT16 unsigned int
```

However, even if you use #defines like these everywhere in your program, there are still some differences between `-L` programs and non–"`-L`" programs. If you are using a 68020, 68030, 68040, or EC–series processor, it is probably best to use the `-L` switch, which makes `int` 32 bits wide. This is a rather subtle point, and depends on close reading of ANSI C.

The biggest different is in parameter passing. In C, all integral arguments smaller than `int` are widened to `int` before they are pushed onto the stack. Thus if you compile with `-L`, then you will always pass integers as 32 bits. If you compile without `-L` then you will pass small integers as 16 bits.

Passing arguments as 16 bits saves some stack space, and makes better code for passing small integer constant arguments. However, it has two distinct disadvantages:

- The stack may become misaligned:

  This is only an issue with the 68020, 68030, 68040, 68060 and the associated EC–series processors. On these processors, a fullword access to a fullword–aligned address executes faster than a fullword accesses to a non–fullword–aligned addresses. If an odd number of parameter words are pushed on the stack, then the next frame will be misaligned, causing inefficient accesses to variables on the stack. This is not an issue for the 68000 or CPU32 processors, because word alignment is just as fast as fullword alignment on those processors.

- Procedure call mismatch errors can occur:

  If you have a procedure which expects a long (32–bit) parameter, and you accidently pass it an argument of type `int`, then only 16–bits will be pushed on the stack if `-L` is not supplied. This would cause the code to execute incorrectly. These errors can be avoided by using function prototypes, of course. However, if you don't always use prototypes, then you must code like this:

```
f ( (INT32)6 );
```

  OR:

```
f ( 6L );
```

Another difference between -L and non-"-L" compilation is the semantics of 16-bit arithmetic. Suppose you add two 16-bit integers and store the result in a 32-bit integer. Under -L, C says that you sign-extend both 16-bit integers to 32 bits and then add them as 32-bit integers. Under non-"-L", C says that you add them as 16-bit integers, and then sign-extend the result to 32 bits. These two sequences give the same answer if the result is in range for a 16-bit integer. If the answer is not in range, then the -L sequence will compute the correct answer and the non-"-L" sequence will truncate.

Generally speaking, the -L sequence is to be preferred, because it always gets the right answer. However, the non-"-L" sequence is more efficient.

Note that is you add two 16-bit integers and store the result in a 16-bit integer, then both the -L and non-"-L" cases would generate the same code: add the two 16-bit integers as 16-bit integers. Any overflow would be truncated away, so this is legal.

In summary:

- The default (not -L) compilation model is likely to be more efficient for 16-bit operations, but it can be more error-prone. Operations done in 16 bits may cause destructive overflow, and more kinds of parameter-argument mismatch errors are possible.
- The -L compilation model is probably better for the 68020, 68030, 68040, 68060 and EC-series processors because it guarantees that the stack will remain fullword aligned.

7.7.5 Compilation Models for Data

The InterTools compiler supports several different strategies for addressing data. Although the default models are usually best, it is sometimes possible to improve efficiency by using a different model.

First, consider data on the stack. Normally, the compiler uses a frame pointer (A6) to address variables on the stack. This straightforward scheme is used by almost all 68000–family compilers. However, it does consumes a valuable register (A6), and requires a relatively expensive instruction pair to set up A6 (LINK/UNLK).

The alternative is to use the stack pointer (A7) to address variables on the stack. This is very much more complicated for the compiler, because A7 moves around every time something is pushed on the stack. It also makes it just about impossible for the debugger to keep track of what is going on, and it makes it harder to read and understand the code. Nevertheless, this does lead to more efficient code. This "no frame pointer" strategy is selected by using the `-n6` switch. Code compiled with `-n6` can be successfully combined with code that is compiled without `-n6`, so it is not necessary to rebuild the runtime library to use `-n6`.

Next, consider the non–stack data. Normally, the compiler uses the "16–bit displacement from A5" addressing mode, "`d16(A5)`", to access non–stack data. Variables marked as "separate", either by the "`#pragma separate`" directive or various compilation options are addressed via 32–bit direct addressing, "`d32`".

The "`d16(A5)`" addressing mode is smaller than the "`d32`" addressing mode, and slightly faster in most cases. However, this strategy does consume the valuable A5 register. The "no–A5" strategy may be very much better if your application spends a lot of time in subroutines that can use the additional A5 register effectively.

The "no–A5" compilation strategy is relatively more effective on the 68040 and the 68EC040 processors. This is because these processors execute the "`d32`" addressing mode at the same speed as "`d16(A5)`", rather than slightly slower. So there is no speed advantage gained by using A5. In fact, there is one case where using "`d32`" over "`d16(A5)`" yields a speed advantage. The most common case is where the address of a non–stack variable is assigned to a pointer which is not in a register. If the variable is separate, then you get this instruction:

```
MOVE.L    #d32,pointer
```

If the variable is A5–relative, then you get these two instructions:

```
LEA.L d16(A5),A0
MOVE.L    A0,pointer
```

On a 68040, the first sequence is faster. On a 68020 they are equally fast. Thus, on a 68040, the "no–A5" strategy can be expected to to run slightly faster than the default "A5" strategy.

To use the "no–A5" compilation strategy, supply `-n5` and one of `-sd`, `-ss`, or `-sc`. This makes all your non–stack data "separate", i.e., directly addressed, and tells the compiler that it may use A5 for other purposes. Note that you must also build yourself a "no–A5" runtime library. This process is described in detail elsewhere in the *InterTools User's Manual.*

Note that the "no–A5" strategy does have other implications. For example, A5–relative data can be dynamically allocated, where separate data must be statically allocated. This consideration may prevent you from using the no–A5 strategy for your application.

7.8 Support for the On–board Peripherals of the 68332, 68340, and 68360

The run–time libraries contain many files to support access, via C or assembly language, to the on–board peripheral units of the 68332, 68340, and 68360 processors. Access to the peripheral components is achieved through the use of C and assembly language include files. These files are listed below:

CPU	Peripheral	C Include File	Assembly Language Include File
68332	System Integration Module	`sim30.h`	`sim30.h68`
68332	Queued Serial Module	`qsm30.h`	`qsm30.h68`
68332	Standby RAM	`ram30.h`	`ram30.h68`
68332	Timer Processor Unit	`tpu30.h`	`tpu30.h68`
68340	System Integration Module	`sim40.h`	`sim40.h68`
68340	Direct Memory Access (DMA) Controller Module	`dma40.h`	`dma40.h68`
68340	Serial Module	`sio40.h`	`sio40.h68`
68340	Timer Modules	`tim40.h`	`tim40.h68`
68360	System Integration Module	`sim60.h`	`sim60.h68`
68360	Communication Processor Module	`cpm60.h`	`cpm60.h68`
68360	Dual–Port RAM	`dpram60.h`	`dpram60.h68`

In the C include files listed above, several C types are used to describe the components of the various peripherals. These C types are defined in the library include file `stypes.h`. You do not need to include this file yourself, because it is included by the C include files listed above. The fundamental types defined in `stypes.h` are `_BYTE` (an unsigned char type), `_WORD` (a 16–bit unsigned integer type), and `_DWORD` (a 32–bit unsigned integer type).

Each C include file above defines a structure whose fields correspond to the particular peripheral unit's components. As an example, to access the Module Control Register (MCR) of the 68332's System Integration Module (SIM), you might have code like the following:

```
#include "sim30.h"
...
if (_SIM.MCR == init_value) {
...
}
```

Each assembly language include file uses `EQU` statements to define the peripheral components as offsets from the peripheral's base address. Thus, to access the Module Control Register (MCR) of the 68332's System Integration Module (SIM) via assembly language, you might have code like the following:

```
include 'sim30.h68'
...
move.w __MCR,d0
...
```

The C include files for the 68340's Serial Module (`sio40.h`) and for the 68360's Dual–Port RAM (`dpram60.h`) are a little more complicated.

In the case of the 68340's Serial Module, there are some components which are read–only, some which are write–only, and some which are read/write. Some of these read–only and write–only components are mapped into the same location. The components which can be both read and written are accessed through the external variable, `_SIO`. Those which are read–only are accessed through either `_SIO` or `_SIOR`; the components which are write–only are accessed through `_SIOW`. For example, the following names all reference a register that is at located 0x11 bytes from the start of the `SIO` module:

```
_SIO.SRA    — Status Register A (read–only)
_SIOR.SRA   — Status Register A (read–only)
_SIOW.CSRA  — Clock Select Register A (write–only)
```

Note, however, that

```
_SIO.CSRA   — Clock Select Register A (write–only)
```

is illegal since `CSRA` is a write–only component; it is not a field of the `_SIO` structure.

In the case of the 68360's Dual–Port RAM, the memory map can be used in different ways so it is defined using multiple levels of

structures and unions. Here are some sample references to components of the Dual–Port RAM:

```
#include "dpram60.h"
 ...
i1  = _DPRAM.USR_DATA[1];      /* User Data / BDs / Microcode        */
                                  /* Program                         */
i2  = _DPRAM.SCC1.UART.TBASE; /* UART-mode SCC1 Tx BD Base          */
                                  /* Address                         */
i3  = _DPRAM.SCC2.HDLC.C_MASK;    /* HDLC-mode SCC2 CRC Constant     */
i4  = _DPRAM.SCC1.BISYNC.BDLE;    /* BISYNC-mode SCC1 BISYNC DLE     */
                                  /* Character                       */
i5  = _DPRAM.SCC1.TRANS.TSTATE;   /* Transparent-mode SCC1 Tx        */
                                  /* Internal State                  */
i6  = _DPRAM.SPI.RBASE;           /* SPI Rx BD Base Address          */
i7  = _DPRAM.TIMER.R_TMR;         /* RISC Timer Mode Register        */
i8  = _DPRAM.IDMA1.ISTATE;        /* IDMA Internal State             */
i9  = _DPRAM.SMC1.UART.TBASE;     /* UART-mode SMC1 Tx BD Base       */
                                  /* Address                         */
i10 = _DPRAM.SMC1.TRANS.RBASE;    /* Transparent-mode SMC1 Rx BD     */
                                  /* Address                         */
i11 = _DPRAM.SMC1.GCI.CI_RxBD;    /* GCI-mode SMC1 C/I Channel Rx
BD*/
...
```

Code which uses any of these C or assembly language include files (and is linked with the appropriate InterTools run–time library) is automatically linked with a target–specific object module which defines a segment containing entry points for that target's peripheral units. For the 68332, this object module is from the run–time library file `mc68332.68k`. For the 68340, the run–time library file is `mc68340.68k`. For the 68360, the file is `mc68360.68k`.

On reset, the 68332 peripheral areas are mapped into 68332 memory starting at address 0xFFFA00. However, by clearing the `MM` bit in the System Integration Module (SIM) Module Control Register (MCR), you can remap them into addresses starting at 0x7FFA00. If you choose to do this, you must also change the absolute location of the `MC68332_SUBSYSTEMS` segment defined in the `mc68332.68k` module.

If the 68340 or 68360 is **NOT** being used with the corresponding debug monitor (e.g., 340bug or 360bug), then the peripheral areas may be mapped into any 4K–byte aligned address. The address of the area used must be written into the Module Base Address Register (MBAR) before the on–chip peripherals can be used. For the 68340, the MBAR is set up by the 68340 version of `pmain`, defined in the file `pmn340.68k`. For the 68360, the MBAR is set up by the 68360 version of `pmain`, defined in the file `pmn360.68k`.

If the 68340 or 68360 is being used with the corresponding debug monitor, the MBAR is already set up by the monitor. Thus, the 68340 and 68360 libraries that should be used with a debug monitor do not set the MBAR. Instead, the `mc68340b.68k` and `mc68360b.68k` files define an absolute segment containing the entry points for the on–board peripherals.

For more information, refer to the run–time library files mentioned above.

7.9 InterTools FLEXlm Guide (Unix only)

This application note discusses Globetrotter Software's Flexible License Manager and how it is integrated into the InterTools Toolkit. It also contains descriptions of the Flexible License Manager license administration tools that are included with the Toolkit, runtime error messages that may occur during operation, the daemon log file and its contents, providing for uninterrupted access to the InterTools Toolkit, and the use of daemon options files to customize your use of the InterTools Toolkit.

7.9.1 License Administration

Overview

The Flexible License Manager (FLEXlm) is a set of software libraries and utilities that, when incorporated into software such as the InterTools Toolkit, provides for managing access to the software. The InterTools software is granted permission to run by FLEXlm daemons; the daemons are started when the InterTools Toolkit is installed and run continuously thereafter. Information needed by the FLEXlm daemons to perform access management is contained in a license data file that is created during the Toolkit installation process. As part of their normal operation, the daemons log their actions in a daemon log file, which can be used to monitor usage of the InterTools Toolkit.

The following sections discuss:

- Installation of the FLEXlm daemons to provide for uninterrupted access to the InterTools Toolkit.
- Customizing your use of the Toolkit through the use of a daemon options file.
- Utilities that are provided to assist you in performing license administration functions.
- The daemon log file and its contents.
- Runtime messages that may occur during operation of the Toolkit.

For additional information regarding the use of FLEXlm, refer to the *Installation Guide* chapter in the *InterTools Getting Started Manual.*

Providing For Uninterrupted FLEXlm Operation

As part of the InterTools Toolkit installation, a *flexlm* subdirectory is created under the root installation directory. This directory contains:

`license.dat`	The license file that was created during installation.
`lmgrd`	The FLEXlm daemon (license daemon).
`IMSI`	The InterTools daemon.
`lm*`	A group of FLEXlm license administration utilities.

As part of the initial InterTools Toolkit installation, the two license management daemons, `lmgrd` and `IMSI`, are invoked and an installation verification script is executed. If the verification is successful, the Toolkit can be used by any user who has defined an environment variable, `LM_LICENSE_FILE`, that defines the full pathname of the license file that was created during installation. For example, if the license file is contained in

/usr/local/itools/flexlm/license.dat, the following command added to a user's .login file would be appropriate:

```
setenv LM_LICENSE_FILE
/usr/local/itools/flexlm/license.dat
```

This particular installation, however, will only be in effect while the FLEXlm daemons continue to run. If the server node on which the daemons were initially installed must be re–booted, the daemons would have to be manually restarted to allow use of the InterTools Toolkit. This process can be automated by providing for the daemons to be started in one of two system start–up files: /etc/rc.boot or /etc/rc.local; assuming that the daemons were installed in /usr/local/itools/flexlm, the following line would be added to the appropriate start–up file:

```
/usr/local/itools/flexlm/lmgrd -c /usr/local/itools/flexlm/ \
   license.dat > /usr/local/itools/flexlm/license.log &
```

This would cause the two daemons to be started, using /usr/local/itools/flexlm/license.dat as the license file and /usr/local/itools/flexlm/license.log as the daemon log file.

Any user who wishes to utilize the license administration utilities (normally, this would be restricted to a system administrator) should add the flexlm installation directory to his execution path; in the above example, this directory is /usr/local/itools/flexlm.

Daemon Options File

It is possible to customize the use of InterTools software using a daemon options file. This options file allows you to reserve licenses

for specified users or groups of users, to restrict access to InterTools, and to set software timeouts. The following keywords are recognized at the start of a line of a daemon options file:

KEYWORDS	FUNCTION
RESERVE	Ensures that InterTools software will always be available to one or more users or on one or more host computer systems.
INCLUDE	Allows you to specify a list of users who are allowed exclusive access to the InterTools software.
EXCLUDE	Allows you to disallow certain people use of the InterTools software.
GROUP	Allows the specification of a group of users for use in the other commands.
TIMEOUT	Allows licenses that are idle to be returned to the free pool, for use by someone else.
NOLOG	Causes messages of the specified type to be filtered out of the daemon's log output.

In order to use the daemon options capability, you must create a daemon options file and list its pathname as the fourth field on the DAEMON line for the IMSI daemon in the license file. A daemon options file consists of lines in the following format:

```
RESERVE     number feature{USER | HOST | DISPLAY |
GROUP} name
INCLUDE     feature{USER | HOST | DISPLAY | GROUP}
name
EXCLUDE     feature{USER | HOST | DISPLAY | GROUP}
name
GROUP       name <list_of_users>
TIMEOUT     feature timeout_in_seconds
NOLOG       {IN | OUT | DENIED | QUEUED}
REPORTLOG   file
```

Lines beginning with the sharp character (#) are ignored, and can be used as comments. For example, the following options file would reserve one copy of feature bc68000 for user "beth", three copies for user "kerry", and one copy for anyone on a computer with the hostname of "lisa"; and would cause QUEUED messages to be omitted from the log file. In addition, user "steve" and group "slackers" would not be allowed to use the feature bc68000:

```
GROUP       slackers jeff jason brandon
RESERVE1    bc68000 USER beth
RESERVE3    bc68000 USER kerry
RESERVE1    bc68000 HOST lisa
EXCLUDE     bc68000 USER steve
EXCLUDE     bc68000 GROUP slackers
NOLOG       QUEUED
```

If this data were in file `/usr/local/local.options`, then you would modify the license file DAEMON line as follows:

```
DAEMON IMSI /usr/local/IMSI
/usr/local/local.options
```

License Administration Tools

The following utilities are provided to facilitate license management by your system administrator. In certain cases, execution access to a utility is restricted to users with root privileges. Complete descriptions of these utilities are provided at the end of this section.

`lmstat`

License administration is simplified by the `lmstat` utility. `lmstat` allows you to instantly monitor the status of all network licensing activities. `lmstat` allows a system administrator to monitor license management operations including:

- Which daemons are running
- Users of individual features
- Users of features served by a specific DAEMON

The usage of `lmstat` is as follows:

```
lmstat    [-a] [-S [DAEMON]] [-f [feature]]
          [-s [server]] [-t value] [-c
license_file]
          [-A] [-l [regular expression]]

-a                          — Display everything
-A                          — List all active licenses
-c license_file             — Use "license_file"
-S [DAEMON]                 — List all users of DAEMON's
features
-f [feature_name]           — List users of feature(s)
-l [regular expression]     — List users of matching license(s)
-s [server_name]            — Display status of server node(s)
-t value                    — Set lmstat timeout to "value"
```

lmdown

The `lmdown` utility allows for the graceful shutdown of all license daemons (both `lmgrd` and all vendor daemons, such as `IMSI`) on all nodes. To use `lmdown`, simply type "`lmdown`" with the correct license file in either `/usr/local/license.dat`, or the license file pathname in the environment variable `LM_LICENSE_FILE`. In addition, `lmdown` takes the "`-c license_file_path`" argument to specify the license file location. Since shutting down the servers will cause loss of licenses, execution of `lmdown` is restricted to users with root privileges.

lmremove

The `lmremove` utility allows the system administrator to remove a single user's license for a specified feature. This could be required in the case where the licensed user was running the software on a node that subsequently crashed. This situation will sometimes cause the license to remain unusable. `lmremove` will allow the license to return to the pool of available licenses.

lmremove is used as follows:

```
lmremove [-c file] feature user host display
```

lmremove will remove all instances of "user" on node "host" on display "display" from usage of "feature". If the optional -c file is specified, the indicated file will be used as the license file. Since removing a user's license can be disruptive, execution of lmremove is restricted to users with root privileges.

lmreread

The lmreread utility will cause the license daemon to reread the license file and start any new vendor daemons that have been added. In addition, all pre–existing daemons will be signaled to re–read the license file for changes in feature licensing information. Usage is:

```
lmreread [-c license_file]
```

Note: If the -c option is used, the license file specified will be read by lmreread, NOT by lmgrd; lmgrd re–reads the file it read originally. Also, lmreread cannot be used to change server node names or port numbers. Vendor daemons will not re–read their option files as a result of lmreread.

7.9.2 FLEXlm User Commands

LMDOWN(1)

NAME

lmdown – graceful shutdown of all license daemons

SYNOPSIS

```
lmdown [ -c license_file ] [ -q ]
```

DESCRIPTION

`lmdown` allows the system administrator to send a message to every license daemon asking it to shut down. The license daemons write out their last messages to the log file, close the file, and exit. All licenses which have been given out by those daemons will rescinded, so that the next time a client program goes to verify his license, it will not be valid.

OPTIONS

`-c` *license_file*
: Use the specified *license_file*. If this switch is not specified, `lmdown` looks for the environment variable `LM_LICENSE_FILE`. If that environment variable is not set, `lmdown` looks for the file `/usr/local/flexlm/licenses/license.dat`. If no `-c` option is specified, `lmdown` looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, `lmdown` looks for the file `/usr/local/flexlm/licenses/license.dat`.

`-q`
: Quiet mode. If this switch is not specified, `lmdown` asks for confirmation before asking the license daemons to shut down. If this switch is specified, `lmdown` will not ask for confirmation.

SEE ALSO

`lmgrd(1)`, `lmstat(1)`, `lmreread(1)`.

LMGRD(1)

NAME

lmgrd – flexible license manager daemon

SYNOPSIS

```
lmgrd [ -c license_file ] [ -t timeout ] [ -s interval ]
```

DESCRIPTION

lmgrd is the main daemon program for the FLEXlm distributed license management system. When invoked, it looks for a license file containing all required information about vendors and features.

OPTIONS

-c *license_file*
: Use the specified *license_file*. If this switch is not specified, lmgrd looks for the environment variable LM_LICENSE_FILE. If that environment variable is not set, lmgrd looks for the file /usr/local/flexlm/licenses/license.dat.

-t *timeout*
: Specifies the *timeout* interval, in seconds, during which daemons must complete their connections to each other. The default value is 10 seconds. A larger value may be desirable if the daemons are being run on busy systems or a very heavily loaded network.

`-s` *interval* — Specifies the logfile timestamp *interval*, in minutes. The default is 360 minutes.

ENVIRONMENT

If no `-c` option is specified, `lmgrd` looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, `lmgrd` looks for the file `/usr/local/flexlm/licenses/license.dat`.

SEE ALSO

`lmdown(1)`, `lmstat(1)`

LMHOSTID(1)

NAME

lmhostid – report the hostid of a system

SYNOPSIS

```
lmhostid
```

DESCRIPTION

lmhostid calls the FLEXlm version of gethostid and displays the results.

The output of lmhostid looks something like this:

```
lmhostid - Copyright (C) 1989, 1991 Highland Software,
Inc.
The FLEXlm host ID of this machine is "1200abcd"
```

OPTIONS

lmhostid has no command line options.

LMREMOVE(1)

NAME

lmremove – remove specific licenses and return them to license pool

SYNOPSIS

```
lmremove [ -c license_file ] feature user host [ display ]
```

DESCRIPTION

lmremove allows the system administrator to remove a single user's license for a specified feature. This could be required in the case where the licensed user was running the software on a node that subsequently crashed. This situation will sometimes cause the license to remain unusable. lmremove will allow the license to return to the pool of available licenses.

OPTIONS

-c *license_file* — Use the specified *license_file*. If this switch is not specified, lmremove looks for the environment variable LM_LICENSE_FILE. If that environment variable is not set, lmremove looks for the file /usr/local/flexlm/licenses/license.dat. If no -c option is specified, lmremove looks for the environment variable LM_LICENSE_FILE in order to find the license file to use. If that environment variable is not set, lmremove looks for the file /usr/local/flexlm/licenses/license.dat.

SEE ALSO

`lmstat(1)`.

LMREREAD(1)

NAME

lmreread – tells the license daemon to reread the license file

SYNOPSIS

```
lmreread [ -c license_file ]
```

DESCRIPTION

lmreread allows the system administrator to tell the license daemon to reread the license file. This can be useful if the data in the license file has changed; the new data can be loaded into the license daemon without shutting down and restarting it.

lmreread uses the license file from the command line (or the default file, if none specified) only to find the license daemon to send it the command to reread the license file. The license daemon will always reread the original file that it loaded. If you need to change the path to the license file read by the license daemon, then you must shut down the daemon and restart it with that new license file path.

You can not use lmreread if the SERVER node names or port numbers have been changed in the license file. In this case, you must shut down the daemon and restart it in order for those changes to take effect.

lmreread does not change any option information specified in an options file. If the new license file specifies a different options file, that information is ignored. If you need to reread the options file, you must shut down the daemon and restart it.

OPTIONS

`-c` *license_file*

Use the specified *license_file*. If this switch is not specified, `lmreread` looks for the environment variable `LM_LICENSE_FILE`. If that environment variable is not set, `lmreread` looks for the file `/usr/local/flexlm/licenses/license.dat`. If no `-c` option is specified, `lmreread` looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, `lmreread` looks for the file `/usr/local/flexlm/licenses/license.dat`.

SEE ALSO

`lmdown(1)`.

LMSTAT(1)

NAME

`lmstat` – report status on license manager daemons and feature usage

SYNOPSIS

```
lmstat [ -a ] [ -A ] [-c license_file ]
    [ -f [feature] ] [ -l [regular_expression] ] [ -s [server] ]
    [ -S [daemon] ] [ -t timeout ]
```

DESCRIPTION

`lmstat` provides information about the status of the server nodes, vendor daemons, vendor features, and users of each feature. Information can be qualified optionally by specific server nodes, vendor daemons, or features.

OPTIONS

`-a`	Display everything.
`-A`	List all active licenses.
`-c` *license_file*	Use the specified *license_file*. If this switch is not specified, `lmstat` looks for the environment variable `LM_LICENSE_FILE`. If that environment variable is not set, `lmstat` looks for the file `/usr/local/flexlm/licenses/license.dat`.

`-f` [*feature*] List all users of the specified *feature*(s).

`-l` [*regular_expression*] List all users of the features matching the given *regular_expression*.

`-s` [*server*] Display the status of the specified *server* node(s).

`-S` [*daemon*] List all users of the specified *daemon*'s features.

`-t` *timeout* Specifies the *timeout* interval, in seconds, during which daemons must complete their connections to each other. The default value is 10 seconds. A larger value may be desirable if the daemons are being run on busy systems or a very heavily loaded network.

ENVIRONMENT

If no `-c` option is specified, `lmstat` looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, `lmstat` looks for the file `/usr/local/flexlm/licenses/license.dat`.

SEE ALSO

`lmgrd(1)`.

7.9.3 FLEXlm Runtime Error Messages and Codes

This section contains a list of possible runtime errors that can occur as the result of FLEXlm license management failure. Each item in the list has the following format:

Code NAME

`error text`

Description

Where each field is defined as follows:

Code	A negative number representing the FLEXlm error that occurred.
NAME	The internal symbolic name corresponding to the error code.
`error text`	The text of the error message that will be displayed to the user.
Description	A brief summary of the possible causes of the error.

–1 NOCONFFILE

`cannot find license file`

The license file cannot be opened. FLEXlm will attempt to open the standard file:

`(/usr/local/flexlm/licenses/license.dat)`

or the file specified by the vendor (via `lm_setup()` **or** `lm_set_attr()`), **or** the file specified by the user in `LM_LICENSE_FILE`.

–2 BADFILE

`invalid license file syntax`

A feature name is > `MAX_FEATURE_LEN`.

A daemon name is > `MAX_DAEMON_LEN`.

A server name is > `MAX_SERVER_NAME`.

A feature specifies no `hostid` and # of licenses is <= 0.

–3 NOSERVER

`cannot connect to a license server`

The daemon name specified in the license file `FEATURE` line does not match the vendor daemon name.

–4 MAXUSERS

`licensed number of users already reached`

The licensed number of users has been reached, and no additional licenses are available.

–5 NOFEATURE

`no such feature exists`

The feature could not be found in the license file.

A call was made to one of the software license working group usage metering functions (`um_xxx_record()`).

–6 NOSERVICE

`no TCP "license" service exists`

This happens if a `SERVER` line does not specify a `TCP/IP` port number, and the TCP license service does not exist in `/etc/services`.

–7 NOSOCKET

`no socket connection to license manager server`

`lm_disconn()` was called after the process had been disconnected from the socket. This error can also occur if an internal error happens within `l_sndmsg()` or `l_rcvmsg()`.

-8 BADCODE

`encryption code in license file is inconsistent`

The code in a license file line does not match the other data in the license file. This is usually the result of not building all the software components with the same encryption code. Check `create_license.c`, `ls_vendor.c`, and your application code carefully to insure that they are all built with the same vendor code.

-9 NOTTHISHOST

`invalid host`

The `hostid` specified in the license file does not match the node on which the software is running.

-10 LONGGONE

`feature has expired`

The feature has expired, i.e., today's date is after the expiration date in the license file.

-11 BADDATE

`invalid date format in license file`

The start or expiration date in the license file is invalid.

–12 BADCOMM

`invalid returned data from license server`

The port number returned from `lmgrd` is invalid.

An attempted connection to a vendor daemon did not result in a correct acknowledgement from the daemon.

The daemon did not send back a message within the timeout interval.

A message from the daemon had an invalid `checksum`.

An `lm_userlist()` request did not receive the correct data.

–13 NO_SERVER_IN_FILE

`no SERVER lines in license file`

There is no `SERVER` line in the license file. All non–zero license count features need at least one `SERVER` line.

–14 BADHOST

`cannot find SERVER hostname in network database`

The `gethostbyname()` system call failed for the `SERVER` nodename in the license file.

–15 CANTCONNECT

`cannot connect to license server`

The `connect()` system call failed, while attempting to connect to the daemon.

The attempt to connect to the vendor daemon on all `SERVER` nodes was unsuccessful.

`lm_status()` returns `CANTCONNECT` if the feature had been checked out but the program is in the process of reconnecting.

If reconnection fails, the final status return is `CANTCONNECT`.

–16 CANTREAD

`cannot read data from license server`

The process cannot read data from the daemon within the timeout interval.

The connection was reset by the daemon (usually because the daemon exited) before the process attempted to read data.

–17 CANTWRITE

`cannot write data to license server`

The process could not write data to the daemon after the connection was established.

–18 NOSERVSUPP

`license server does not support this feature`

The feature has expired (on the server node), or has not yet started, or the version is greater than the highest supported version.

–19 SELECTERR

`error in select system call`

The `select()` system call failed. This should never happen in a client program.

–20 SERVBUSY

`license server busy (no majority)`

The license server is busy establishing a quorum of server nodes so that licensing can start. This request should be retried.

–21 OLDVER

`license file does not support this version`

The version requested is greater than the highest version supported in the license file `FEATURE` line.

–22 CHECKINBAD

`feature checkin failure detected at license server`

The `checkin` request did not receive a proper reply from the vendor daemon (the license might still be considered in use).

–23 BUSYNEWSERV

`license server temporarily busy (new server connecting)`

The vendor daemon is in the process of establishing a quorum condition. New requests from clients are deferred during this period. This request should be retried.

–24 USERSQUEUED

```
users are queued for this feature
```

This error is similar to MAXUSERS, but supplies the additional information that there are other users in the queue for this feature.

–25 SERVLONGGONE

```
license server does not support this
version of this feature
```

The version specified in the checkout request is greater than the highest version number the daemon supports.

–26 TOOMANY

```
request for more licenses than this
feature supports
```

A checkout request was made for more licenses than are supported in the license file. This request will never succeed.

–27 CANTREADKMEM

```
cannot read /dev/kmem
```

The file /dev/kmem is not readable on this system. This error will only occur if using extended hostid checking.

–28 CANTREADVMUNIX

```
cannot read /vmunix
```

The file /vmunix is not readable on this system. This error will only occur if using extended hostid checking.

–29 CANTFINDETHER

`cannot find ethernet device`

The ethernet device could not be located on this (VMS) system.

–30 NOREADLIC

`cannot read license file`

The license file could not be read (`errno == EPERM` or `EACCES`).

–31 TOOEARLY

`feature not yet available`

The feature is not enabled yet (current date is before the feature start date).

–32 NOSUCHATTR

`No such attribute`

A call to `lm_get_attr()` or `lm_set_attr()` specified an unknown attribute code.

–33 BADHANDSHAKE

`Bad encryption handshake with daemon`

The client performs an encryption handshake operation with the daemon prior to any licensing operations. This handshake operation failed.

–34 CLOCKBAD

`Clock difference too large between client and server`

The date on the client system does not agree closely enough with the date on the server (daemon) system. The amount of difference allowed is set by the software vendor with `lm_set_attr` by modifying the `LM_A_MAX_TIMEDIFF` parameter.

–35 FEATQUEUE

`In the queue for this feature`

This checkout request has resulted in the process being placed in the queue for this feature. Subsequent calls to `lm_status()` will yield the status of this queued request.

–36 FEATCORRUPT

`Feature database corrupted in daemon`

The daemon's runtime feature data structures have become corrupted. This is an internal daemon error.

–37 BADFEATPARAM

`Duplicate selection mismatch for this feature`

The checkout request for this feature has specified a duplicate mask that does not match the mask specified by an earlier checkout. This is probably the result of using different versions of your client software.

–38 FEATEXCLUDE

`User/host on EXCLUDE list for feature`

The `user/host/display` has been excluded from this feature by an end–user's daemon option file.

–39 FEATNOTINCLUDE

`User/host not on INCLUDE list for feature`

The `user/host/display` has NOT been included in this feature by an end–user's daemon option file.

–40 CANTMALLOC

`Cannot allocate dynamic memory`

The `malloc()` call failed to return sufficient memory. This can happen while reading the features from the license file (especially if the user has an extremely long `LM_LICENSE_FILE` path set).

–41 NEVERCHECKOUT

`Feature was never checked out`

This code is returned by `lm_status()` if the feature requested has never been checked out.

7.9.4 The Daemon Log File

The FLEXlm daemons all generate log files containing messages in the following format:

mm/dd hh:mm (DAEMON *name*) *message*

Where:

mm/dd hh:mm	Is the month/day hour:minute that the message was logged.
DAEMON *name*	Either "license daemon" or the string from the DAEMON line that describes your daemon. In the case where a single copy of the daemon cannot handle all of the requested licenses, an optional "_" followed by a number indicates that this message comes from a forked daemon.
message	The text of the message.

The log files can be used to:

1. Inform you when it may be necessary to update your application software licensing arrangement.
2. Diagnose configuration problems.
3. Diagnose daemon software errors.

The messages are grouped below into the above three categories, with each message followed by a brief description of its meaning.

Informational Messages

`Connected to node`

This daemon is connected to its peer on node *node*.

`CONNECTED, master is name`

The license daemons log this message when a quorum is up and everyone has selected a master.

`DEMO mode supports only one SERVER host!`

An attempt was made to configure a demo version of the software for more than one server host.

```
DENIED: N feature to user (mm/dd/yy hh:mm)
```

user was denied access to N licenses of *feature*. This message may indicate a need to purchase more licenses.

```
EXITING DUE TO SIGNAL nnn
EXITING with code nnn
```

All daemons list the reason that the daemon has exited.

```
EXPIRED: feature
```

feature has passed its expiration date.

```
IN: feature by user (N licenses) (used:
d:hh:mm:ss)
   (mm/dd/yy hh:mm)
```

user has checked back in N licenses of *feature* at *mm/dd/yy hh:mm*.

```
IN server died: feature by user (number
licenses)
   (used: d:hh:mm:ss) (mm/dd/yy hh:mm)
```

user has checked in N licenses by virtue of the fact that his server died.

```
License Manager server started
```

The license daemon was started.

`Lost connection to host`

A daemon can no longer communicate with its peer on node *host*, which can cause the clients to have to reconnect, or cause the number of daemons to go below the minimum number, in which case clients may start exiting. If the license daemons lose the connection to the master, they will kill all the vendor daemons; vendor daemons will shut themselves down.

`Lost quorum`

The daemon lost quorum, so will process only connection requests from other deamons.

`MASTER SERVER died due to signal nnn`

The license daemon received fatal signal *nnn*.

`MULTIPLE xxx servers running. Please kill, and restart license daemon`

The license daemon has detected that multiple copies of vendor daemon *xxx* are running. The user should kill all *xxx* daemon processes and re–start the license daemon.

`OUT: feature by user (N licenses) (mm/dd/yy hh:mm)`

user has checked out `N` licenses of *feature* at *mm/dd/yy hh:mm*

`Removing clients of children`

The top–level daemon logs this message when one of the child daemons dies.

`RESERVE feature for HOST name`
`RESERVE feature for USER name`

A license of *feature* is reserved for either user *name* or host *name*.

`RESTarted xxx (internet port nnn)`

Vendor daemon *xxx* was restarted at internet port *nnn*.

`Retrying socket bind (address in use)`

The license servers try to bind their sockets for approximately 6 minutes if they detect *address in use* errors.

`Selected (EXISTING) master node`

This license daemon has selected an existing master (node) as the master.

`SERVER shutdown requested`

A daemon was requested to shut down via a user–generated kill command.

`[NEW] Server started for: feature-list`

A (possibly new) server was started for the features listed.

`Shutting down xxx`

The license daemon is shutting down the vendor daemon *xxx*.

`SIGCHLD received. Killing child servers`

A vendor daemon logs this message when a shutdown was requested by the license daemon.

`Started name`

The license daemon logs this message whenever it starts a new vendor daemon.

`Trying connection to node`

The daemon is attempting a connection to *node*.

Configuration Problem Messages

`hostname: Not a valid server host, exiting`

This daemon was run on an invalid hostname.

`hostname: Wrong hostid, exiting`

The hostid is wrong for *hostname*.

`BAD CODE for feature-name`

The specified feature name has a bad encryption code.

`CANNOT OPEN options file "file"`

The options file specified in the license file could not be opened.

`Couldn't find a master`

The daemons could not agree on a master.

`license daemon: lost all connections`

This message is logged when all the connections to a server are lost, which often indicates a network problem.

`lost lock, exiting`
`Error closing lock file`
`Unable to re-open lock file`

The vendor daemon has a problem with its lock file, usually because of an attempt to run more than one copy of the daemon on a single node. Locate the other daemon that is running via a `ps` command, and kill it with `kill -9`.

`NO DAEMON line for daemon`

The license file does not contain a `DAEMON` line for *daemon*.

`No "license" service found`

The TCP *license* service did not exist in `/etc/services`.

`No license data for "feat", feature unsupported`

There is no feature line for *feat* in the license file.

`No features to serve!`

A vendor daemon found no features to serve. This could be caused by bad data in the license file.

`UNSUPPORTED FEATURE request: feature by user`

The *user* has requested a feature that this vendor daemon does not support. This can happen for a number of reasons: the license file is bad, the feature has expired, or the daemon is accessing the wrong license file.

`Unknown host: hostname`

The hostname specified on a `SERVER` line in the license file does not exist in the network database (probably `/etc/hosts`).

`lm_server: lost all connections`

This message is logged when all the connections to a server are lost. This probably indicates a network problem.

`NO DAEMON lines, exiting`

The license daemon logs this message if there are no `DAEMON` lines in the license file. Since there are no vendor daemons to start, there is nothing to do.

`NO DAEMON line for name`

A vendor daemon logs this error if it cannot find its own `DAEMON` name in the license file.

Daemon Software Error Messages

`accept: message`

An error was detected in the `accept` system call.

`ATTEMPT TO START VENDOR DAEMON xxx with NO MASTER`

A vendor daemon was started with no master selected. This is an internal consistency error in the daemons.

`BAD PID message from nnn: pid: xxx (msg)`

A top–level vendor daemon received an invalid `PID` message from one of its children (daemon number *xxx*).

`BAD SCONNECT message: (message)`

An invalid "server connect" message was received.

`Cannot create pipes for server communication`

The `pipe` call failed.

`Can't allocate server table space`

A `malloc` error. Check swap space.

`Connection to node TIMED OUT`

The daemon could not connect to *node*.

`Error sending PID to master server`

The vendor server could not send its `PID` to the top–level server in the hierarchy.

`Illegal connection request to DAEMON`

A connection request was made to `DAEMON`, but this vendor daemon is not `DAEMON`.

`Illegal server connection request`

A connection request came in from another server without a `DAEMON` name.

`KILL of child failed, errno = nnn`

A daemon could not kill its child.

`No internet port number specified`

A vendor daemon was started without an internet port.

`Not enough descriptors to re-create pipes`

The "top–level" daemon detected one of its sub–daemon's death. In trying to restart the chain of sub–daemons, it was unable to get the file descriptors to set up the pipes to communicate. This is a fatal error, and the daemons must be re–started.

`read: error message`

An error in a `read` system call was detected.

`recycle_control BUT WE DIDN'T HAVE CONTROL`

The hierarchy of vendor daemons has become confused over who holds the control token. This is an internal error.

`return_reserved: can't find feature listhead`

When a daemon is returning a reservation to the "free reservation" list, it could not find the `listhead` of features.

`select: message`

An error in a select system call was detected.

`Server exiting`

The server is exiting. This is normally due to an error.

`SHELLO for wrong DAEMON`

This vendor daemon was sent a "server hello" message that was destined for a different `DAEMON`.

```
Unsolicited msg from parent!
```

Normally, the top–level vendor daemon sends no unsolicited messages. If one arrives, this message is logged. This is a bug.

```
WARNING: CORRUPTED options list (o->next == 0)
Options list TERMINATED at bad entry
```

An internal inconsistency was detected in the daemon's option list.

APPENDIX A

C Language Specifications

This appendix discusses preprocessor extensions, in–line assembly, ANSI C function prototypes, the const type qualifier, the volatile type qualifier, and implementation–defined behavior.

1 Introduction

The InterTools C language consists of ANSI standard C, plus some extensions described in this appendix. This appendix also focuses on some of the new parts of ANSI C and how they interact with various compiler options. The final section describes how some aspects of ANSI C that are "left to the implementation" have been done by the InterTools compiler.

There are some aspects of ANSI C which are only supported in the presence of compiler options. This is done in the interest of backwards compatibility and code quality. For example, ANSI C requires that the size of the "short" datatype to be at least 16 bits long. By default the InterTools 68000compiler maps the "short" datatype into an 8–bit integer. Therefore the InterTools compiler is not ANSI compliant unless one of `-L` (16–bit shorts, 32–bit ints) or `-D s2s` (16–bit shorts) is supplied. There are two other cases discussed below where the default behavior of the compiler does not agree with the ANSI standard, but these are unlikely to affect most users. See the *C Compiler* chapter for more details.

ANSI C permits floating point expressions to be computed in greater precision than their types would indicate, but it does not permit floating point variables to be stored in greater precision than their type. This requirement has no special implications in the software floats case. However, in the hardware floats case this requirement makes floating point register variables illegal, since 68881/68882 floating point registers hold 80–bit extended precision values. Strict adherence to this rule is quite expensive in code quality terms, so it is only provided if the `-sp` switch is supplied.

Finally, ANSI C permits redundant declarations of the form:

```
int i;
int i = 1;
```

to appear within a single module. Since the InterTools compiler segregates initialized from uninitialized data via the `idata/udata` segments, this requirement forces the compiler to delay emitting data allocation statements for uninitialized variables until the end of the compilation. This requirement slows down the compiler and has the possibly undesirable side effect of emitting uninitialized declarations in a different order than their original declarations.

This change should not affect legal C programs, but in practice it may cause programs to execute differently. For this reason this kind of declaration is only permitted if the `-dd` switch is supplied.

Note that these two declarations would not be legal if they were compiled in separate modules and linked together. This is not a violation of the ANSI C standard. The InterTools compiler implements the "strict def–ref model" of external names. Every external name must have exactly one module containing a "defining declaration", i.e., one without the `extern` keyword. All other modules must contain only "referencing declarations", i.e., ones with the `extern` keyword. In that spirit, duplicate declarations of the form permitted by the `-dd` switch are considered bad form but legal.

2 Preprocessor Extensions

The InterTools C preprocessor is functionally equivalent to the standard ANSI C preprocessor, and includes the additional features listed in the table below:

Command	Function	Default
#list on	List following lines until `#list off` (or EOF)	list on
#list off	Disable listing until `#list on` (or EOF)	
#list page	Start new page; print page header	
#list skip *n*	Put *n* blank lines in listfile	*n* = 1
#list title *string*	*string* becomes the page header. Also does a `#list page` command	null title
#pragma separate... #pragma sep_on #pragma sep_off	See the *Pragma Separate (Option Separate)* application note	

Note: The `#pragma separate` statements can be equivalently coded as `#option separate`.

3 In–line Assembly Language

In–line assembly language is a language extension which permits assembly language code to be inserted into the code generated by the C compiler. This is similar to calling an out–of–line routine coded in assembly language except that the call and return overhead is eliminated.

Normally the InterTools compiler produces an object module directly rather than generating assembly language and assembling it into an object module. However, when in–line assembly language is present, an assembly step must be used to process the inserted assembly language source. The `-ia` command line option directs the compiler to generate assembly language rather than an object module, and then to invoke the assembler to generate the final object module. This switch makes the compiler run more slowly, but it is required when in–line assembly language insertions are used.

The simplest form of in–line assembly is provided by the `_ASMLINE` built–in function. By coding:

```
_ASMLINE("string");
```

you cause the compiler to emit the indicated line directly into the generated assembly language (the compiler will append a newline character). This construct may appear within or between procedures. Ordinary C escape processing is **NOT** performed on the assembly language string, so you can't embed newlines using `\n`. Also, remember that in assembly language only labels can begin in column 1, so you probably will want to put a leading blank in your string. The resulting code is otherwise ignored by the compiler.

Many other compilers support this feature using `asm` in place of `_ASMLINE`. The InterTools compiler uses `_ASMLINE` because the ANSI C standard says that C programs should to be able to use `asm` as an identifier. If you prefer `asm`, either add "`#define asm _ASMLINE`" to your program or use the equivalent command line switch, "`-P asm=_ASMLINE`".

This feature is very simple and straightforward, but it has several weaknesses. First, it cannot receive arguments or return results to the surrounding C code. Second, because it is completely ignored by the optimizer, it must not cause side–effects (such as modifying registers and non–volatile global variables) which could invalidate the results of optimization. In cases where more flexibility is needed, a pre–defined in–line assembly insertion must be used.

An in–line assembly language insertion is defined as a kind of pseudo–function whose body is coded in assembly language. This pseudo–function is called an "asm macro." An assembly language macro declaration is preceded by keyword `_CASM` or `_ASM`, and its body is coded in assembly language, but otherwise it obeys the same C syntax rules as an ordinary function declaration. It may have a function prototype. There are two restrictions: `_ASM` macros may not have more than 16 parameters, and neither `_ASM` nor `_CASM` macros may return a value of type aggregate or (except under the hardware floating point option) double.

The two forms, `_CASM` and `_ASM`, reflect two alternate methods of parameter passing. In the `_CASM` method, parameter setup is performed exactly as it would be before an out–of–line call. That is, the parameters are evaluated one by one and pushed onto the stack. In the `_ASM` method, parameter setup is performed by a different method which is intended to minimize the parameter setup code. The exact method used is described below in more detail. In general, the `_CASM` method is easier to use, but the `_ASM` method is more powerful and may lead to more efficient object code.

An assembly language macro is invoked by using the ordinary C syntax for a procedure call, giving the name of the assembly language macro as the function being "called." The compiler replaces the call instruction (JSR or BSR) that would be generated with the body of the assembly language macro.

Here is an example:

```
_CASM void disable_interrupts() {
    or  #$0700,sr
}
_CASM void enable_interrupts() {
    and #$f8ff,sr
}
f() {
    disable_interrupts();
    ...
    enable_interrupts();
}
```

This example shows two assembly language macros. One of them enables interrupts and the other disables interrupts, by changing the interrupt priority mask in the status register. The "call" to `disable_interrupts` causes the "or to status register" instruction to be inserted into the body of the procedure `f` immediately after the prologue sequence. The "call" to `enable_interrupts` causes the "and to status register" instruction to be inserted into the body of the procedure `f` immediately before the epilogue sequence. Therefore the procedure `f` would run with interrupts disabled.

Generally speaking, all the rules for interfacing C and assembly language apply equally well to assembly language macros. In particular, the same set of registers can be modified and return values are transmitted in the same way. See the *Linking C and Assembly* application note for a detailed explanation of these rules. In fact, assembly language routines originally coded to be called from C can be converted to `_CASM` assembly language macros in a very straightforward way.

In–line assembly language must be used carefully, especially if the optimizer is being used. The optimizer assumes that in–line assembly language insertions are entered at the top and exit (if at all) at the bottom. In particular, an in–line assembly insertion must not contain jumps out of that insertion and into another insertion. It also must not write into the stack storage belonging to the "calling" routine.

An assembly language macro represents a sequence of instructions that are repeated at every call. It therefore provides an in–line procedure facility. As with any kind of in–line procedure facility, this can represent a space/time tradeoff: there is no call overhead, so it is faster, but the entire body is repeated each time, so it may be larger. An assembly language macro thus has no address, and so it cannot be invoked through a "pointer to procedure" variable as a normal subroutine can.

A detailed description of the syntax of in–line assembly macro definitions appears at the end of this section. Next, we will describe the two methods, `_CASM` and `_ASM`, and discuss their differences.

3.1 The _CASM method

The `_CASM` method is the most straightforward method of in–line assembly. As was mentioned above, the code generated for a call to a `_CASM` macro is exactly the same as would be generated for an out–of–line call. Therefore the parameters will be in a predictable place. On entry to the macro, the first parameter will be on top of the stack (pointed to by A7). Note that an out–of–line routine would find its return address on top of the stack, and the first parameter four bytes higher up.

Here is an example of a simple assembly language routine coded to be called from C:

```
    section  S_f,,"code"
    xdef  _setvbr
_setvbr  equ   *
    move.l   4(a7),d0
    movec.l  d0,vbr
    rts
    end
```

This routine expects one 4–byte parameter, and stores that parameter into the VBR register. Its entry point is `_setvbr`, and it is stored in a segment named `S_setvbr`. This definition is consistent with this external C declaration:

```
extern void setvbr(long);
```

Here is the definition for an equivalent `_CASM` assembly language macro:

```
_CASM void setvbr(long value) {
    move.l   (a7),d0
    movec.l  d0,vbr
}
```

Note that the parameter is now expected at `(a7)` rather than `4(a7)` because there will be no "return address" on the stack. Also, note that the `rts` (return from subroutine) instruction is absent. The external label `_setvbr` has also been removed, as has the section directive and the end directive. Here is an invocation of this macro.

```
 t() {
setvbr(256);
 }
```

Here is the code that would result from the call:

```
pea.l     256
move.l    (a7),d0
movec.l   d0,vbr
addq.l    #4,a7
```

The first instruction is the parameter setup for the "call," the next two are the body of the assembly language macro, and the last instruction is the stack cleanup that normally follows a call. It pops off the parameters that were pushed for the call.

3.2 The _ASM method

The _ASM method is designed to allow the most efficient object code to be generated. It not only eliminates the call and return overhead, it also eliminates the stack cleanup after the call, and usually all of the parameter setup code as well. On the other hand, it is harder to use.

The example shown in the _CASM example in the previous section shows the inefficiencies imposed by a standard calling convention. The _ASM method attempts to eliminate parameter setup code by adopting the convention that parameters are passed "where they are." For example, if the actual parameter was the name of a global variable, then at a _CASM call it would be pushed on the stack, while at an _ASM call it would stay where it is. That is, no setup code would be emitted.

This leads to the first question: how can the body of the assembly language macro refer to its parameters? The answer is that the body of an _ASM macro references its parameters using the syntax of 68000 family assembly language macros. That is, \1 refers to the first parameter, \2 to the second, and so on.

This is not enough to solve all the problems in referencing parameters. Sometimes it is necessary to know whether a parameter is a constant, a memory location, or a register. For example, references to constants must be preceded by a pound sign (#) in 68000 family assembly language. Also, some instructions require register operands, while others accept either register or memory operands. For example, the first operand of the `MOVEC` instruction must be a register.

This second problem is solved by having alternative expansions of the macro for different kinds of actual parameters. Here is an example showing the same `setvbr` example from the `_CASM` section recoded as an `_ASM` macro:

```
_ASM void setvbr(long value) {
%reg value
   movec.l  \1,vbr
%con value
   move.l   #\1,d0
   movec.l  d0,vbr
%mem value
   move.l   \1,d0
   movec.l  d0,vbr
}
```

Here the `%reg value` line is a predicate which delimits the first alternative expansion of the `setvbr` assembly language macro. This expansion consists of all the assembly language lines up to the next predicate, marked by `%con value`. The second alternative expansion consists of all lines up next predicate, `%mem value`. The third alternative expansion consists of all lines up to the right curly brace that terminates the assembly language macro definition.

Note: If you wish to have a null body predicate, you must insert a blank line or a comment between successive predicates. Otherwise, the compiler will treat the adjacent "%" lines as one predicate definition.

The compiler chooses one alternative expansion at each call, depending on the form of the actual parameters at that call. If the actual parameter was a register variable, then the first expansion, `%reg`, would be chosen. If the actual parameter was a constant, then the second expansion, `%con`, would be chosen. If the actual parameter was a memory location, then the third expansion, `%mem` would be chosen. There is another predicate, `%var`, which was not used here, which matches parameters of type "variable," that is, register or memory but not constant.

If the argument of an `_ASM` macro cannot be characterized as a register variable, a constant, or a memory variable, then the compiler generates a temporary variable, copies the actual parameter into the temporary, and passes the temporary in its place. Memory locations accessed via pointers or arrays subscripted by non–constant subscripts are also handled in this way.

Here is an invocation of this assembly language macro.

```
t() {
setvbr(256);
}
```

Here is the code that would result from the call:

```
move.l    #256,d0
movec.l   d0,vbr
```

Note that the constant parameter, 256, is not pushed on the stack before the macro invocation, nor is it popped afterwards. This saves two instructions over the _CASM method. On the other hand, the source for the _CASM form of the setvbr macro was four lines long, while the _ASM form is ten lines long, because it is necessary to code three alternative expansions. This is a fairly typical tradeoff.

When an _ASM macro has more than one parameter, the predicates become more complex. It is then necessary to specify the characteristics of all the parameters in combination. For two parameters, this could require nine alternatives (three possible types, register, constant, or memory for both parameters). If it can be limited to two types (constant or variable), then it would require only four alternatives. For three parameters, it could require between eight and twenty–seven alternatives. Clearly, it is impractical to use the _ASM method with assembly language macros that have large numbers of parameters.

Note: An _ASM macro must not modify its parameters. This violates the assumptions of the compiler, and may result in incorrect optimization. For example, the compiler may load a local variable into a register before an assembly language macro, pass that variable to an assembly language macro, and then use that register after the assembly language macro. Therefore an assignment to that variable within the assembly language macro would not have the intended effect. An assembly language macro may modify global variables and may modify the targets of pointer variables. If you want to modify a local variable in an assembly language macro, you should pass its address to the macro, not its value.

An _ASM macro can freely use the scratch registers, D0, D1, A0, A4, FP0, and FP4, without having to save and restore them. If the -ar compiler switch is used, then the _ASM macro can instead use scratch registers D0, D1, A0, A1, FP0, and FP1. In either case, it may also use other registers, as long as it saves them and restores them before exiting. Note however that register parameters may be resident in these other registers. Therefore, before overwriting any other registers, care should be taken to make sure that no parameter could be in that register. For example, suppose an _ASM macro wants to make use of the D2 register. If there are any parameters of integral type, then one might be in D2 at some invocation of this macro. The prudent course would be to move the parameter elsewhere, say, D0, before pushing D2 and overwriting its value.

3.3 Syntax Summary

Having given examples of each form, here is the syntax of the _CASM and_ASM macros:

```
_CASM <C-style function header> {
    asm-line
    ...
}
```

```
_ASM <C-style function header> {
    predicate
    asm-line
    ...
    predicate
    asm-line
    ...
...
}
```

```
_ASM <C-style function header> {
    asm-line
    ...
}
```

Here *asm–line* is a complete line of assembly language, terminated by a newline. It may contain any characters, but the first non–white–space character may not be a pound sign (#), a percent sign (%), or a right curly brace (}).

The term *predicate* is a complete line with the following syntax:

```
%mem|reg|con|var parameter [;|,parameter]...

      or

%always
```

The percent sign must be the first non–white–space character on the line. It is followed by a list of pairs consisting of one of the four keywords, `mem`, `reg`, `con`, or `var`, followed by a comma–separated list of formal parameter names. Each predicate must name all of the formal parameters exactly once. For readability the elements of the list may be separated by commas or semicolons. Long predicates may be continued on successive lines, as long as each begins with a percent character.

Lines whose first non–white–space character is a pound sign are treated as preprocessor directives and are interpreted in the normal way. In particular, it is possible to use conditional inclusion (`#if ... #endif`) inside macro definitions. However, the *asm–line*s inside the macro are not scanned by the compiler. Thus, a preprocessor macro inside an *asm–line* would **not** be expanded.

The right curly brace that terminates the body of the `_ASM` or `_CASM` macro must be the first non–white–space character on that line.

The body of an _ASM macro is divided into a list of alternatives, each consisting of a *predicate* and a list of *asm–line*s. When an _ASM macro is invoked, the compiler chooses one these alternatives by comparing the arguments at that invocation with the *predicate*s. Each *predicate* that matches the arguments is chosen. The "always" *predicate* always matches, regardless of the arguments.

Note: The macro body must follow the rules, syntax, and label restrictions described in the *Macros Operations and Conditional Assembly* chapter in the *InterTools Reference Manual*.

A reg condition matches if the argument is contained in one of the following registers:

- D2 – D7
- A1 – A3
- FP1 – FP3
- FP5 – FP7

A con condition matches if the argument is a constant.

A mem condition matches if the argument is a memory location whose address is either a 32–bit constant, a 16–bit constant plus the A5 register, or a 16–bit constant plus the A6 register.

A var condition is equivalent to either reg or mem being true.

The compiler ensures that each argument at an _ASM macro invocation matches one of these cases. If necessary, code will be generated to move the argument into a temporary variable. Note that the compiler may allocate variables and common subexpressions into registers, so it is not always obvious which alternative will be chosen at a particular invocation.

For example, an argument like `a+b` might be a register if the optimizer determined that the value `a+b` was useful later on in the program.

4 ANSI C Function Prototypes

Function prototypes, as described in the ANSI C standard, are supported by InterTools compilers. Function prototypes are function declarations and function definition headers that include a list of the datatypes of the function's parameters. They are used to ensure that all calls, declarations, and definitions of the function that are within the scope of the prototype contain the declared number, type and order of arguments or parameters.

Function prototypes may appear in two contexts: in the headers of function definitions, and in function–type declarations. Although function prototypes are quite useful, they are not required. The following description briefly summarizes the use of function prototypes. Consult an ANSI C reference manual for more details.

4.1 Creating Function Prototypes

In an old–style definition for a function — without a function prototype — the function header contains a parenthesized list of parameter names, followed by parameter declarations.

For example:

```
   /* Not a function prototype */
int func (param1, param2)
   int * param1;
   char param2;
{
   /* function body */
}
```

An old–style declaration for this function could be:

```
int func(); /* Not a function prototype */
```

This syntax has been expanded for the creation of function prototypes. In function definitions that act as function prototypes, a parenthesized, comma–separated, list of parameter types and identifiers *replaces* the list of parameter names and the parameter type declarations of the old–style function definition header. As an example, the function defined above could have been defined to include a function prototype in its header:

```
   /* function prototype */
int func (int * param1, char param2)
{
   /* function body */
}
```

Note that the parameter declarations are now incorporated into the function prototype on the first line of the function definition.

Function declaration syntax has also been expanded. In function declarations that will act as prototypes, the function name is followed by a parenthesized, comma separated list of parameter types and optional identifiers.

A declaration for the function defined above, which will act as a prototype for the function, could be:

```
extern int func (int * x, char y);
```

or

```
extern int func (int *, char);
```

These two declarations are equivalent; the identifiers *x* and *y* are ignored.

The types and numbers of the parameters listed in a function prototype declaration should match the types and numbers of parameters in the function definition (prototype or old–style), but the identifiers used in the declaration prototype declaration need not match the definition.

Function prototypes may be written for functions that take a variable number of arguments. The ellipsis notation, ", ...", used as the last element in a parameter type list in a function prototype, indicates that an unspecified number of arguments follow. At least one parameter must precede the ellipsis in the function declaration. For example, if a function is declared as:

```
int func1 (char *fmt, int num, ...);
```

it may be called with two or more arguments.

A function prototype with a parameter type list that consists solely of the keyword `void` is used to declare a function that has no parameters.

For example, the function

```
int func2 (void);
```

has no parameters. Invoking this function with any arguments would be incorrect.

Clearly, it is desirable to construct include files that use prototype–style declarations for global subroutines. To facilitate this process, use the make prototypes switch, `-mp`. This automatically generates a header file containing prototype declarations for all the subroutines defined in the module being compiled. See the *C Compiler* chapter for more details.

4.2 Calls to Functions with Prototypes

The number of arguments in a function call must correctly correspond to the number of parameters in an in–scope function prototype. If a function's prototype has a `void` type list, there must be no arguments in the call. If the ellipsis notation was used to define a prototype for a function that takes a variable number of parameters, the function call should contain at least as many arguments as there were parameters before the ellipsis. For all other functions declared with prototypes, the number of arguments at the call should match the number of parameters in the prototype.

If a function prototype is in scope of a function call, the arguments are converted, as if by assignment, to the types of the corresponding prototype parameters. If the prototype for a function used the ellipsis notation, this conversion is done only for parameters that were explicitly declared in the prototype; after the ellipses, the default C promotions (including widening of float to double) are done.

Note that the promotion of float to double is not done if a prototype is in scope. This provides a way for an actual float value to be passed to a function. However, it does mean that if a function defined with a prototype has a float parameter, then a protoytpe must be in scope at every call. In fact, ANSI C formally requires that a prototype be in force at every call to any procedure defined with a prototype. This is a good practice, but is not actually required unless a float parameter is involved.

Programmers familiar with strongly–typed languages such as Pascal or Ada are often surprised that no warning is given at an apparent type mismatch between the actual and formal parameters. In ANSI C, prototypes cause *conversions*, not checks. For example, if an integer is passed to a routine whose prototype calls for a double, than an integer to double conversion would occur prior to the call, just as it would at an assignment.

InterTools C compilers emit error messages when function declarations, calls or definitions are incompatible with function prototypes that are in scope.

There is one technicality in the ANSI C prototype rules which confuses many people, so we will describe it thoroughly here. Suppose you have a compilation which contains both prototype and old–style declarations for the same procedure. ANSI C requires that they be compatible with one another. The tricky part is the way this compatibility is checked.

Generally speaking, the rule is that the default C promotions are performed on the non–prototype side before making the comparison. This means that this natural looking program is in error:

```
extern void f1(char);
void f1(c)
   char c;
{
   /* function body */
}
```

The problem here is that the default C promotions turn `char` into `int`, and `int` is not compatible with `char`. This program **IS** correct:

```
extern void f1(int);
void f1(c)
   char c;
{
   /* function body */
}
```

If you think about it, this does make sense. The underlying assumption is that a procedure defined in the old style expects to be called in the old way, i.e., with no prototype in force at the call. When no prototype is in force the default C promotions are applied to the arguments. Therefore an old–style definition really expects to receive its arguments in promoted form. A prototype, on the other hand, expects to receive its parameter in whatever way is most efficient for its type.

As a matter of fact, the InterTools compiler does pass `char` parameters as `int`, even if a prototype is present. This is a small loss in efficiency, but it is much less error–prone. In the case of float versus double however there really is a difference; old–style definition routines expect double, while prototype definition routines expect float.

5 Other ANSI C Features

This section contains descriptions of:

- Adjacent String Literal Concatenation
- Trigraph Replacement
- `void` Pointers
- `const` Type Qualifier
- Stringization
- `volatile` Type Qualifier
- Preprocessor Additions

6 Adjacent String Literal Concatenation

In the ANSI C standard, adjacent strings are automatically concatenated, with a single null character appended to the end of the resulting string.

Example

```
char test[] =  "This "
          "is a "
          "test";
```

This is the equivalent to the following assignment:

```
char test[] = "This is a test";
```

This addition makes it unnecessary to use the line continuation convention to write very long string constants.

7 Trigraph Replacement

Trigraphs let you write C programs on computers using a subset of the ASCII character set. Trigraphs are introduced by two consecutive question marks. The only legal trigraphs are:

trigraph	is equal to
??(	[
??)	]
??<	{
??>	}
??/	\
??'	^
??=	#
??–	~
??!	|

A new escape character (\?) prevents the translation of trigraph–like constructs. For example:

trigraph form	string
"Eh\?\?!"	"Eh??!"
"Backslash is ??/"	"Backslash is \ "

8 Void Pointers – void *

The ANSI C standard states that the type `void *` will be the generic pointer type. Pointers can be assigned into and from void pointers silently and without casting. A void pointer may not be dereferenced without an explicit cast.

Example

```
void * f1;
int * i1;
int i2 = 5;

i1 = &i2;
f1 = i1; /* silent assignment into a void pointer
*/

printf("The integer in f1 "
    "is %d0",*(int *)f1);  /* example of */
              /* neccessary cast */
```

9 Const Type Qualifier

Another new feature in ANSI C is the `const` keyword. This keyword is used to define a read–only type. Note that `const` can be used both

with object declarations and with pointer types. Here are some examples:

```
const double pi = 3.14159;
const double *ptr_to_const;
double d;
double *const const_ptr = &d;
double *non_const_ptr;

/* the following are now illegal */
pi = 3.14;              /* assigns to constant */
const_ptr = &d;         /* assigns to constant */
*ptr_to_const = 3.14; /* assigns to constant */
non_const_ptr = &pi;   /* ptr type mismatch */

/* these are legal */
ptr_to_const = &pi; /* ptr types match */
*const_ptr = 3.14; /* value pointed to by a
constant
                      ptr is non-const */
ptr_to_const = &d; /* ptr to const can point to
                      non_const */
```

The compiler ensures that objects with `const`–qualified types are not modified. Objects declared with `const`–qualified types must therefore be initialized. Global variables declared with `const`–qualified type must be declared `extern const` to prevent other modules from modifying the object.

There are two different ways to use the `const` keyword. The most typical use is for true constants, such as those which might be located in ROM. However, the `const` qualifier can also be used to control access to sensitive variables. For example, it is possible to declare a variable without the `const` keyword, and then provide an external declaration which declares it `extern const`. This technique prevents code outside the defining module from modifying the variable. `const` thus provides support for information management.

If the only use of `const` is for true constants, then it is possible to direct the compiler to segregate all `const` variables into a separate segment named `cdata` of class `{constant}`. This can be done by using the `-cs` compiler option. This makes it easier to locate all `const` variables in ROM storage.

The `-cs` option has the effect of adding an implicit `#pragma` directive of the following form for each `const` variable.

```
#pragma separate my_var segment cdata class
constant
```

Note that the `-cs` switch must **not** be used if `const` is being used for information management. This is because the generated code will not operate correctly if there is a mismatch between the non–separate definition and the implicit external separate declaration created by the option. This is a special case of the general rule that separate variables must be declared external separate and non–separate variables must be declared external non–separate.

10 Stringization

Within macro definitions, the # character is recognized as a unary "stringization" operator that has to be followed be a formal parameter name. When macro expansion occurs, the # and formal name are replaced by the corresponding actual argument enclosed in string quotations. Any double quotes (") and backslashes (\) are automatically escaped with a preceding backslash.

For example, the following source text:

```
#define STRING(a,b) printf(#a ", "#b)

STRING(hello,world);
```

will become, after stringization

```
printf("hello""","""world");
```

which will, after string concatenation, become

```
printf("hello, world");
```

11 ANSI C Preprocessor Additions

The following sections describe valid ANSI C preprocessor additions.

11.1 New Predefined Macros

There are three new predefined macros, defined by the draft proposed ANSI C standard. These macros are:

__STDC__	for an ANSI C conforming compiler will equal 1. The InterTools C compiler will define this as zero until the compiler achieves full ANSI compatibility.
__DATE__	is the date of the compilation. It is set once and does not change, regardless of the compilation length. The format of the date is *MMM DD, YYYY*, where days less than 10 are indicated by a space followed by the day. *MMM* represents the month in alphabetic characters.
__TIME__	is the time of the compilation. It is set once during compilation and does not change, regardless of the compilation length. The format of the time is *HH:MM:SS*.

11.2 New Directives

The ANSI C standard defines three new preprocessing directives: `#error`, `#pragma` and `#elif`.

11.3 #error

The format for `#error` is:

`#error` *errmsg*

The directive causes a compiler error with the given *errmsg* printed out.

11.4 #pragma

`#pragma` is a synonym for `#option`, and supports the same syntax. Any other parameter to `#pragma` which would not be legal with `#option` will give an "Unknown pragma" warning.

11.5 #elif

The `#elif` directive acts like a combination of `#else` and `#if`. The `#elif` comes between `#if` and `#endif`, and has a constant expression to be evaluated in the same way as `#if`. The use of this directive allows for a simpler syntax as the following example shows:

Directive	Becomes
`#if`	`#if`
`#else` `#if`	`#elif`
`#endif` `#endif`	`#endif`

12 Volatile Type Qualifier

The ANSI C `volatile` keyword is used to allow higher levels of optimization without adversely affecting programs which use memory–mapped I/O and interrupt processing, as many embedded applications do.

Many optimizations rely on tracking what a program is doing and looking for a more efficient way to get the same result. This kind of analysis is much more effective when the optimizer can make certain common–sense assumptions based on the way that computer memory works. In particular, it is natural to assume that two successive loads from the same memory location without any possible intervening store must yield the same value. Using this principle, an optimizer may eliminate the second load, thus improving the efficiency of the generated code.

This reasonable–sounding assumption can be violated in two ways. First, if the memory location represents a memory–mapped I/O port, then successive loads correspond to successive read operations. Second, an asynchronous interrupt handler could modify the memory location between the two load operations. Variables whose values may change without any apparent cause are called "volatile."

If the optimizer can be told which variables are volatile, then it can selectively optimize the non–volatile variables more aggressively. This is the motivation for the `volatile` keyword.

The `volatile` keyword can be used to qualify any C type. It can also be used with pointer types. Here are some examples:

Use the `volatile` keyword as follows. First, decide which variables in your program are volatile. This includes any memory–mapped I/O ports and any variables modified by interrupt handlers. Mark these objects as volatile. Next, determine which pointer variables are used to access these volatile objects, and mark their pointed–to type with the `volatile` keyword. Be sure to watch for pointers formed by type–casting constants into addresses.

If you do not feel confident that you can locate and appropriately qualify all your volatile variables, then you can avoid inappropriate optimizations by using the `-vv` switch. Note, however, that this switch may make the generated code significantly larger and slower.

13 New Operators

13.1 defined

In an `#if` there is a new operator: `defined`. This operator returns true if its parameter is a currently defined macro that has not been subject to an `#undef`.

The syntax is as follows:

`#if defined` (*identifier*)

or

`#if defined` *identifier*

This is like `#ifdef` and `#ifndef`, but it returns a boolean so expressions can be created, like the following:

```
#if defined (macro_one) && !defined(macro_two)
    .
    .
    .
#endif
```

13.2 token pasting

Another new operator is the token pasting operator (##). This is used inside macro definitions to concatenate two tokens, to create a new token. For example:

```
#define pasting(x,y) x ## y

int pasting(x,1);
```

would become:

```
int x1;
```

after macro substitution and expansion.

14 Support for Interrupt Handlers in C

Special interrupt services are provided by the InterTools C compiler as extensions to the language. Three pseudo–functions and two function–type keywords are defined:

- `_GPL()`
- `_SPL(`*n*`)`
- `_TRAP(`*n*`)`
- `_IH`
- `_SWI`

These features will be described in detail below.

The complete description of exception processing is beyond the scope of this manual, but is fully described in the User's Manual for the microprocessor. First, we will briefly review the general nature of exception processing to describe how the compiler can be used to code handlers for exceptions, generate traps and manipulate the status register.

Exception processing may be initiated in several different ways, which fall into the following general categories:

- `TRAP` instruction
- Interrupt by external device
- Instruction trace, i.e., machine single step
- System error, e.g., bus error or divide by zero

When an exception occurs, some information is stored on the stack and execution passes through an exception vector. The exception vectors reside at fixed offsets from the base of the exception vector table. The particular vector chosen depends on the kind of exception. The exception vector table is located at address 0 for the MC68000, but for other M68000 family processors its address is defined by the VBR register.

A routine which receives control after an exception is called a *handler.* It takes whatever action is appropriate for the given exception. When it finishes, the handler may return control to the routine which was active when the exception occurred. This kind of return must be done via the `RTE` (return from exception) instruction. The `RTE` pops off the information which was stored on the stack when the exception occurred. It then restores the program counter and status register to their value at the time of the exception.

With the InterTools 68000 Family Toolkit, you can use the `_IH` or `_SWI` keywords to designate a given C function as an exception handler. This causes the compiler to generate an `RTE` instruction in the epilogue of the function instead of the usual `RTS` return instruction. Furthermore, the compiler will ensure that the function's prologue preserves the entire machine state, not just those registers which, by convention, are preserved across ordinary function calls. The difference between `_IH` and `_SWI` is described in detail below.

A routine designated as a handler must have the `void` return type and may not take any parameters. It must not be called as a procedure by any other code. Like any normal function, handlers may declare local data, access global data, and call other functions. If you are using the `volatile` keyword, please note that global variables which are modified by exception handlers must usually be designated as `volatile`.

Note that the appropriate exception vector must be initialized in order to establish a given function as a handler for a particular exception. This can be done with "ordinary" C code: typecasting the vector's address into "pointer to pointer to function" provides a pointer to the vector. This pointer can be used to initialize the vector with the address of the handler. Thus, no special language extension is required.

External interrupts have one of eight different priority levels. The status register contains a 3–bit field which defines the interrupt priority mask. Interrupts of priority less than or equal to the current interrupt priority mask level are postponed until the mask becomes low enough to unblock the interrupt. Interrupts of priority seven are a special case; they may not be inhibited by the priority mask, thus providing a non–maskable interrupt. When an interrupt occurs the interrupt priority mask is set to the level of the interrupt being serviced. The handler's `RTE` instruction restores the status register, thus restoring the interrupt priority mask.

The `_GPL` and `_SPL` pseudo–functions allow the user to read and write the interrupt priority mask. The `_TRAP` pseudo–function generates a `TRAP` instruction.

14.1 The _GPL Pseudo–Function

`_GPL` returns the value of the interrupt priority mask in the status register. The result is an integer whose value is between zero and seven.

14.2 The _SPL Pseudo–Function

`_SPL`(*n*) sets the value of the interrupt priority mask in the status register according to the low order three bits of the value of **n**. The sequence for `_SPL(5)` looks like this:

```
MOVE    SR,D1          get status register
AND     #$F8FF,D1      clear the mask field
OR      #$500,D1       set the mask field
MOVE    D1,SR          update status register
```

_SPL returns the new value of the interrupt priority mask as a result. If this value will not be used, code to determine it is not emitted.

14.3 The _TRAP Function

_TRAP(*n*) simply forces the compiler to emit the TRAP #*n* instruction. Here *n* must be a constant between zero and fifteen.

14.4 The _IH Keyword

_IH specifies that the defined function is an exception handler. The keyword must precede any class or type information associated with the function. The type of any _IH function must be void, and an _IH function may not receive any parameters.

When the compiler recognizes a function as an exception handler, it emits the RTE instruction for returns, instead of the usual RTS instruction. Also, any registers modified by the function will be saved and restored, not just the set of registers designated as preserved across normal function calls. However, registers not used by the function might not be explicitly saved and restored in the function's prologue and epilogue, respectively.

An interrupt handler routine may only be called from C with the _TRAP mechanism since a standard procedure call does not have the same effect as a TRAP instruction.

Example

Definition of interrupt handler in C:

```
 /* _IH KEYWORD EXAMPLES */

int GotInterrupted = 0;

_IH void myhandler () {
   GotInterrupted = 1;
}
```

Example

```
      XREF --main, -myhandler
      ORG 0                        ;assume EVT at 0
   SECTION vectors
      DC.L $00007ffc               ;vector 0
      DC.L  --main                 ;vector 1
      DC.L  -myhandler             ;vector 2
            .
            .
            .
```

Example

```
/* _IH KEYWORD EXAMPLES */

int GotInterrupted = 0;
int Icount;

_IH void myhandler () {
   GotInterrupted = 1;
}

void set_handler(f, vec_number)
   void (*f) ();
   int vec_number;
{
   /* make the vector at address 4*vec_number */
   /* point to the handler procedure f. */
   *((void (**) () )(4*vec_number)) = f;
}
```

14.5 The _SWI Keyword

The _SWI (software interrupt) keyword is similar to the _IH keyword except that it is used for exception handlers which might provoke a context swap. In order to provide for this, the _SWI function explicitly performs a full context save upon entry.

The keyword must precede any class or type information associated with the function. The type of any _SWI function must be void, and an _SWI function may not take any parameters.

Example

```
/* _SWI KEYWORD EXAMPLE */

extern struct TCB {
   int priority;
   int * saved_context;
} *current_task, *highest_prio_task;
extern void swap_process ();

_SWI void timer_handler () {
   if (current_task->priority <
      highest_prio_task->priority) {
      swap_process ();
   }
}
```

15 Implementation–Defined Behavior

The ANSI standard allows each C compiler to behave differently in a fixed set of situations. The behavior of the InterTools C compiler in these situations is described below. Section numbers refer to the ANSI standard document.

(sect. 3.3.3.4, 4.1.1) The type of the `sizeof` operator, `size_t`, is `unsigned long int`.

(sect. 3.3.6, 4.1.1) The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t`, is `long int`.

(sect. 3.5.2.1) A "plain" `int` bit–field is treated as an `unsigned int` bit–field. To make a signed bit–field, use `signed int` type.

(sect. 4.5.1) The mathematics functions return undefined values on domain errors.

(sect. 4.5.1) The mathematics functions set the integer expression `errno` to the value of the macro ERANGE on overflow range errors.

(sect. 4.5.6.4) When the `fmod` function has a second argument of zero, a domain error occurs, and the function returns an undefined value.

(sect. 4.9.6.1) The output for `%p` conversion in the `fprintf` function is as if the conversion specification were `%lx`.

(sect. 4.9.6.2) The output for `%p` conversion in the `fscanf` function is as if the conversion specification were `%lx`.

(sect. 4.9.6.2) A – (hyphen) character in any position in the scanlist for a `%[]` conversion in the `fscanf` function is treated like any other character. For example, `%[a-z]` will match any sequence of `a`, `-`, or `z` characters. It will not match any sequence of lower case alphabetic characters.

(sect. 4.10.3) The functions `calloc` and `malloc` return NULL if the size requested is zero. The function `realloc` frees the memory specified and then returns NULL if the size requested is zero.

The ANSI standard does not specify the method by which two declarations of the same external name are combined at link time. The InterTools compiler implements the most common method of resolving this problem, called the "strict def–ref" or "omitted–extern" solution.

In the strict def–ref model, declarations with the `extern` keyword are considered "referencing" declarations. A declaration without the `extern` or `static` keyword that appears outside of any function is considered a "defining" declaration. The InterTools compiler requires that there be exactly one module which contains a defining declaration for each external name. All other modules may contain only referencing declarations for that external name.

A new switch has been added to the compiler to allow ANSI–style duplicate declarations. By default the InterTools compiler gives an error if more than one definition for a variable is present in a single module. The effect of the `-dd` switch is to permit multiple definitions in a single module, as required by ANSI C. Note that this switch changes the order of allocation for uninitialized global variables. Programs that depend on global variables being allocated in the order they are declared must not use this switch.

APPENDIX B

Compiler Naming Conventions

This section describes Code Symbols, Data Symbols, and Segment Names and contains a Symbol Naming Summary.

1 Introduction

This section describes the naming conventions of the C68000 compiler system. Familiarity with the compiler's naming conventions will make it easier to:

- Read the pseudo–assembly listing from the compiler.
- Write linking locator commands to control placement of code and data into target memory.
- Read the listing from the global symbol mapper.
- Interface compiled code with assembly language.

Interfacing compiled code with assembly language requires more detailed technical information. See the *Compiler Run Time Conventions* appendix.

The compiler forms linker symbols from the names in a user program. It creates three kinds of symbols:

- Internal symbol names.
- Global symbol names.
- Segment names.

Internal symbols are temporary names which are eliminated as the linking locator processes the object module. They are used to resolve internal references. They are not visible to other modules and thus cannot be referenced by assembly language code. They can be displayed by the symbol list utility, but not by the global symbol mapper. However, they do appear in the pseudo–assembly listing.

Global symbols are visible to other modules, and can be used by assembly language code to reference compiler–generated code and data. They can also be referenced by other compiled modules. They can be displayed with the global symbol mapper.

Segments represent relocatable blocks of target memory. They contain the generated machine instructions and data to be loaded into the target. Segments have a name and are assigned a class and, optionally, a group. See the *Linking Concepts* section in the *Linking Locator* chapter for a more detailed description of segments, classes, and groups.

2 Code Symbols

The compiler creates symbols for procedure entry points. The symbol names are based on the names the user specified in the source program, with an underscore ("_") added to the beginning of the name.

Global functions are those which are visible outside their compilation unit. In C, all functions are global by default. Local functions are indicated with the `static` keyword.

The compiler creates a global symbol for each global function. Its name is formed by prepending an underscore, "_", to the source name. For example, the global function *visible* would generate the global symbol *_visible*. The compiler creates an internal symbol for each local function. It's name is generated using two underscores followed by an "N", a unique number, a period and the class name, in this case "`code`". For example, a local function might generate the name `__N14.code`.

3 Data Symbols

There are several kinds of data which can be defined in a C program. They are:

- **Global Data** – all data visible to other compilation units, i.e., all data declared outside procedure blocks, except that with the `static` attribute.
- **Local Static Data** – all data declared with the `static` attribute.
- **Activation Record Local Data** – formal procedure parameters and data defined inside of procedures, except that with the `static` attribute.

3.1 Global Data

The compiler generates global symbols for all global data items. The symbol name is formed by prepending an underscore, "`_`", to the source name. For example, a global variable whose source name is *global_int* gives rise to a global symbol named *_global_int.*

3.2 Local Static Data

Artificial names are generated for all static variables declared within procedures to avoid name conflicts. The internal symbol name is generated using two underscores and "N" followed by a unique number, a period and the class name if it is not "data". For example, a local static variable might generate a name such as `__N4.myclass`.

3.3 Stack Data

The compiler generates no symbols for stack data. These variables do not have permanent memory locations allocated to them; they are allocated on the run–time stack.

Stack variables are addressed at a constant offset from the stack frame pointer register, A6.

3.4 String Constants

String constants arise from quoted string literals in the source. The compiler generates dummy internal symbol names.

3.5 Other Symbols

The C++ compiler creates symbols with the prefixes `__TIR__`, `__CBI__`, and `__DNI__` for its own internal use.

4 Segment Names

4.1 Code Segment Names

All code in a single module is allocated in one code segment. The segment name is formed by prepending an "S" to the *symbol* name given by the compiler or assembler to the *first* subroutine encountered in the source module. Thus, if a global function named `sort` is the first function in a module, the code generated for *all* procedures in the module will be allocated in a segment named `S_sort`. If a static function named `hidden` is the first function, the generated code will be allocated in a segment named *S__N#.code*, where # is some unique number. The code segment is assigned class {code}, unless the `-cc` switch is specified. The `-cc` switch allows you to set the class name. The *Linking Locator* chapter summarizes the class names.

4.2 Data Segment Names

There are three data segments which are always created by the compiler, regardless of the input program. They are `idata`, `udata`, and `sdata`. `idata` contains initialized data and `udata` contains uninitialized data. Both `idata` and `udata` are assigned group data and class {`data`}. `sdata` contains string constants and is assigned class {constant}. All non–stack data will be in one of these segments unless the `separate` option is used.

The linking locator creates a global symbol named `ldata`, whose value is the size of the data group (`idata` and `udata` segments). Note that ldata is not a segment, but a global symbol. This may be useful to programs which dynamically allocate their global data area.

4.3 Separate Data

Segments are also created whenever data items are declared `separate`. The segment names are either created from the name of the data item or are explicitly specified by the user. In addition to segments, `separate` data items are assigned a class, either by default or by explicit user request. For a detailed explanation of `separate` data, see the *Pragma Separate (Option Separate)* application note.

If neither segment nor class specifications are explicitly given, the segment names for `separate` data are formed by prepending an "S" to the corresponding symbol name. The class name is either {isep}, {usep}, or {stsep}, depending upon whether the `separate` data item is global initialized, global uninitialized, or static.

User Specified		Data	Initial	Resulting	
Segment	Class	Type	Value	Segment	Class
none	none	global	No	S_X	usep
none	none	global	Yes	S_X	isep
none	none	static	NA	S__N#.stsep	stsep
sname	none	global or local	NA	*sname*	separate
none	*cname*	global	NA	S_X	*cname*
none	*cname*	local	NA	S__N#.*cname*	*cname*
sname	*cname*	global or local	NA	*sname*	*cname*

For example, in a global context, the sequence

```
#pragma separate var1
int var1;
```

would produce a segment named *S_var1* of class {usep}. If *var1* were initialized, the class would be {isep}.

In another example, the sequence

```
#pragma sep_on segment myseg
int var2;
```

would produce a segment named *myseg* of class {`separate`}. For this example, initialization of *var2* or the local/global context would have no effect on the segment and class names.

In a third example, the sequence

```
#pragma sep_on segment seg1 seg2 class cl1 cl2
int var3, var4 = 1;
```

would produce a segment named *seg1* of class {*cl1*}, which would hold the initialized data, and a segment named *seg2* of class {*cl2*}, which would hold the uninitialized data, as explained in the *Pragma Separate (Option Separate)* application note.

As a final example, the sequence

```
#pragma sep_on segment seg3 default class cl3
default
int var4 = 3, var5;
```

would produce a segment named *seg3* of class {*cl3*} to hold the initialized data, and a segment named *S_var5* of class {usep} to hold the uninitialized data.

Note: It is possible to create errors by inconsistent assignment of class names to a single segment. If this is done in a single compilation, the compiler will detect the error. If there is a conflict between separately compiled modules, the linking locator will inform you of this discrepancy.

5 Symbol Naming Summary

The following table summarizes the symbol names generated by the compiler. Given an item *X* in the source, here are the names and attributes of the compiler–generated symbols and segments.

If X is	Separate	Initial	Name	Attributes
Global Routine			_*X*	Global Symbol
Global Routine			S_*X*	Segment, class {code}
Local Routine			_*X*	Internal Symbol
Local Routine			S_*X*	Segment, class {code}
Global Variable	Either	Either	_*X*	Global symbol
Local Variable	Either	Either	__N#.stsep	Internal Symbol
Global Variable	No	No	udata	Segment, class {data}
Global Variable	No	Yes	idata	Segment, class {data}
Global Variable	Yes	No	S_*X*	Segment, class {usep}
Global Variable	Yes	Yes	S_*X*	Segment, class {isep}
Local Variable	No	No	udata	Segment, class {data}
Local Variable	No	Yes	idata	Segment, class {data}
Local Variable	Yes	Either	S__N#.stsep	Segment, class {stsep}
String Constant			__N#	Internal Symbol
String Constant			sdata	Segment, class {constant}
Stack Variable				No symbols

5.1 Notes

- The name of the code segment is determined by the first subroutine. Each compilation generates only one code segment.
- Static variables declared inside procedures get dummy internal symbol names.
- Segments `idata` and `udata` belong to group "data."
- This table assumes that no class or segment specifications were used for separate data.

APPENDIX C

Compiler Runtime Conventions

This section describes Storage Allocation, the Segmentation Model, Register Usage, Subroutine Linkage, Stack Layout, and Initial Startup.

1 Introduction

This appendix describes the compiler's code generation conventions. The stack layout is described, as well as the procedure linkage conventions and data allocation rules. The information in this section is primarily intended for those attempting to interface compiled code with assembly language. It will also be useful to those users who are debugging at the machine instruction level, since it will help you follow what the compiler–generated code is doing.

Throughout this appendix, we assume that you are familiar with the information in the *Compiler Naming Conventions* appendix.

2 Storage Allocation

The basic C datatypes are implemented as follows:

`char`	8 bits, unsigned
`short`	8 bits, signed
`int`	16 bits, signed
`unsigned`	16 bits, unsigned
`long`	32 bits, signed
`float`	32 bits
`double`	64 bits
`pointer` (address)	32 bits (absolute address)

2.1 Notes

- Basic datatypes bigger than a byte are aligned on a 16–bit boundary. Datatypes no bigger than a byte are aligned on a byte boundary.
- Each element of a structure or array is aligned therein according to its own alignment. The alignment of an aggregate datatype is defined to be the strictest alignment of any subcomponent. These rules can be modified with the `-pack` compiler switch.
- Successive bits in a bit field are allocated from left–to–right, that is, *most* significant bit first. If the layout of a field would cross two consecutive 16–bit boundaries, the bit field is aligned at the next 16–bit boundary. Bit fields which are completely contained in a byte contribute byte alignment to the surrounding structure. Other bit fields contribute 16–bit alignment to the surrounding structure. These rules can be modified by various compiler switches.
- Some of the above data sizes can be changed with the `-D` compiler switch. In particular, the `-L` switch makes `short` 16 bits and `int` 32 bits.

3 Segmentation Model

Note: In the following sections, the references to the floating point registers apply only under the `-h` hardware floating point option.

User variables are allocated storage in one of the following places:

1. The runtime stack.

2. The A1, A2, A3, D2, D3, D4, D5, D6, D7, FP1, FP2, FP3, FP5, FP6 and FP7 registers.

3. The global data area (idata and udata segments) referenced by A5.

4. Separate segments.

Variables declared in procedure blocks, including formal procedure parameters, are allocated on the runtime stack. The optimizer allocates variables and temporaries into registers. If the optimzer is suppressed, then variables declared with the `register` keyword are allocated in registers. Registers A1–A3 are available for pointers; registers D2–D7 are available for character or integer types; registers FP1–FP3 and FP5–FP7 are available for types `float` or `double`. Variables named in a `#pragma separate` directive are allocated in separate segments. See the *Pragman Separate (Option Separate)* application note for more detail. All other variables are placed in the global data area. Initialized variables are placed in idata uninitialized varialbes in udata.

The compiler addresses variables on the stack using constant offsets from the frame pointer, i.e., the A6 register. Positive offsets indicate references to formal parameters; negative offsets indicate references to local variables. See the *Stack Layout* section below for a picture of the stack.

The total size of the local variables for any single subroutine is limited to 32K. If a single procedure allocates too much data, the compiler will flag a fatal error:

```
source error: local variables
require too much space (fatal)
```

The compiler addresses variables in the global data area using 16–bit offsets from the dedicated register A5. This addressing scheme allows the compiler to address global data very efficiently, since it can use 16–bit offsets rather than full addresses. On the other hand, this implies a system–wide limit of 64K on the size of the global data area.

If you declare too much global data, the compiler will not be affected, but the linking locator will issue an error:

```
group {data} exceeds maximum size
```

If the 64K limit is exceeded, some variables must be made separate to bring the total size below 64K. Variables allocated in separate storage are not restricted in size. Variables in separate storage are addressed using full 32–bit addresses.

4 Register Usage

The compiler reserves the following machine registers for use:

Register	Use
A1, A2, A3	Pointer **register** variables
A5	Pointer to global data area
A6	Frame pointer
A7	Stack pointer
D2, D3, D4, D5, D6, D7	Integer **register** variables
FP1, FP2, FP3, FP5, FP6, FP7	Floating point **register** variables
A0	Pointer return values
D0	Integer return values
FP0	Floating point return values

5 Subroutine Linkage

5.1 Preserved Registers

Every procedure is responsible for preserving the following registers: D2, D3, D4, D5, D6, D7, A1, A2, A3, A5, A6, A7, FP1, FP2, FP3, FP5, FP6, FP7. This rule also applies to any assembly language routines called from compiled code.

5.2 Register Return Values

The compiler expects function return values in registers under the following circumstances:

- Pointer values are returned in **A0**.
- If the hardware floating point option `-h` **is** selected and the software floating point compatibility option `-68` is **not** selected, then `float` and `double` values are returned in **FP0**. Otherwise `float` values are returned in **D0** and `double` values are returned in a temporary stack location.
- Return values of integral type are returned in **D0**.

5.3 Parameter Passing

All parameters are pushed as one (or more) 16–bit word(s). This means, for example, that one–byte structure parameters occupy two bytes on the stack. The contents of the other (high order) bytes are undefined. For more information on function prototypes, see the *C Language Specifications* appendix.

5.4 Calling Sequence

The generated code for a procedure call has the following form:

1. Determine if the function return value will be returned in a register. If not, allocate space for a function return temporary on the stack.
2. Push the arguments onto the stack. The arguments are pushed as words in reverse order, i.e., the last argument is pushed first.
3. If a function return temporary was allocated, push its address.
4. Call the function.
5. Pop the arguments off the stack.
6. If a function return temporary was allocated, deallocate it after it is used.

5.5 Procedure Prologue

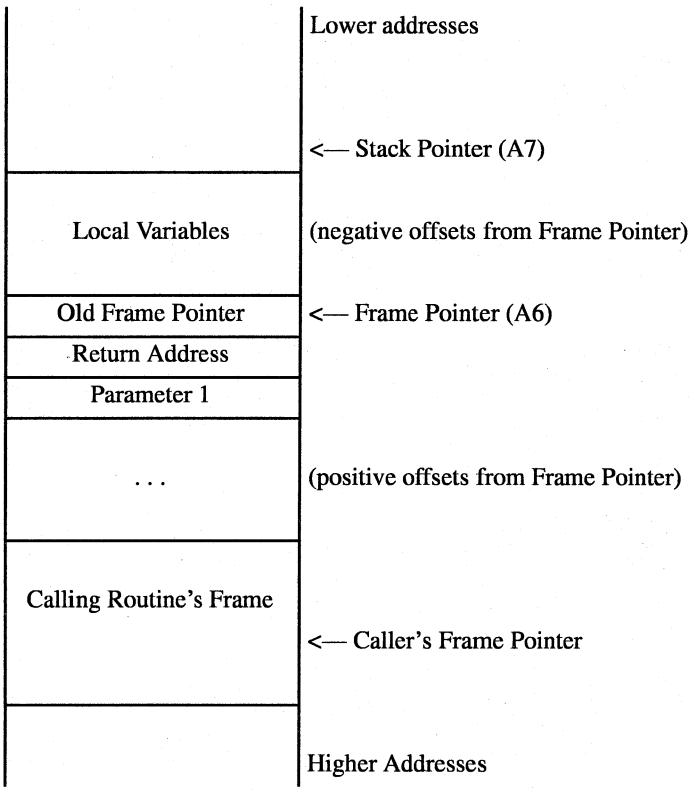
There are three instructions which may appear in the prologue code. The presence of each depends on the nature of the routine. If there are local variables on the stack (or if the `-nl` compiler switch is present) then there will be a `LINK` instruction. If any non–floating point register variables are used by the routine a `MOVEM` is executed. If any floating point register variables are used by the routine an `FMOVEM` is executed. The form of these instructions follows:

`LINK`	`A6,#`*n*	*n* is the size of the new frame
`MOVEM`	*reg list*`,-(A7)`	Save A/D registers
`FMOVEM`	*freg list*`,-(A7)`	Save float registers

Note: The reg list names all the preserved registers, A1, A2, A3, D2, D3, D4, D5, D6, D7, which are modified in the subroutine. The freg list names all the preserved floating point registers, FP1, FP2, FP3, FP5, FP6, FP7, which are modified in the subroutine.

FMOVEM	(A7)+,<freg list>	Restore float registers
MOVEM	(A7)+,<reg list>	Restore A/D registers
UNLK	A6	Restore previous stack frame
RTS		Return to caller

Here is a picture of a typical runtime stack configuration:



5.6 Initial Startup

The compiled code which first receives control from an operating system, executive, power–up sequence, etc., will require certain preparations in order to execute properly. Generally the main requirements are to establish an initial stack frame and to initialize the A5 register with the address of the global data area. Please refer to the *Segmentation Model* subsection above for an explanation of the use of "ldata" in initialization of the A5 register.

A prototype startup routine for use with a ROM–based monitor is provided with the runtime library. This `__main` routine is discussed in detail in the *Runtime Library* chapter in the *InterTools Reference Manual.*

Note: The compiler generates an external reference to the `__main` library routine if the module being compiled contains a routine named main. This allows automatic loading of `__main` from a link library.

APPENDIX D

Object Module Formats

This appendix contains the following sections:

- Introduction
- Intel ASCII Hex Format
- Motorola S Records
- Extended Motorola S Records
- Packed Motorola S Record
- S37 Motorola S Records
- Tektronix Format (Tekhex)
- Extended Tekhex Format
- Binary Tektronix Format
- HP64000 Format
- Common Object File Format (COFF)

1 Introduction

Once a C program has been compiled, linked and located, a file exists that contains all the information required to specify exactly where in memory each part of the program should reside.

However, there is no one standard that determines how this information should be supplied to various PROM burners, emulators, and so on. Each system has its own requirements for file formats, header information, checksums, and other essential information.

InterTools supports a wide variety of ASCII hex and binary object module formats, which are briefly described here. Use the formatter's `-f` switch to choose an output format. For further information on format specifications please refer to the manufacturer's specification for the equipment to which you plan to transfer data.

2 Intel ASCII Hex Format

The general format of a record, shown here with spaces separating each field, is:

```
: ll aaaa tt dd...dd cc
```

In this format field:

:	is the keyword used to signal the start of the record.
ll	is the number of code/data bytes in the record.
aaaa	is the lower 16 bits of the absolute address at which the first byte of code/data in the record is to be placed. (For record types 01 to 03, this field contains "0000". See below.)
tt	represents the record type.
dd..dd	is the data for each record type.
cc	is the checksum.

Record Types:

The following is a list of possible record types (the *tt* field) with the corresponding value of the *ll* field:

tt	*ll*
00 – data record	actual data length
01 – end of file record	00
02 – extended address	02
03 – start address record	04

Record Type Data:

For each record type, the data is as follows:

tt	*dd...dd*
00	the code/data bytes
01	none
02	4 hex digits – the first is the upper four bits of the 20 bit absolute address, followed by 3 zeroes
03	CS and IP (8 digits)

3 Motorola S Records

The general format of a record, shown here with spaces separating each field, is as follows:

ss ll aaaa dd...dd cc

Where:

ss	is the S–record type.
ll	is the record length, which includes the number of bytes in the address, code/data and checksum fields.
aaaa	is the 2–byte address at which the first byte of code/data in the record is to be placed.
dd...dd	is the code/data bytes.
cc	is the checksum

The following is a list of possible S–record types for the *ss* field:

S0 –	header record for each block
S1 –	record containing code/data and 2–byte address at which the code/data is to reside
S9 –	termination record

4 Extended Motorola S Records

The general format of a record, shown here with spaces separating each field, is as follows:

ss ll aaaaaa dd...dd cc

Where:

ss	is the S–record type.
ll	is the record length, which includes the number of bytes in the address, code/data and checksum fields.
aaaaaa	is the 3–byte address at which the first byte of code/data in the record is to be placed.
dd...dd	is the code/data bytes.
cc	is the checksum.

The following is a list of the possible S–record types for the *ss* field:

S0 –	header record for each block.
S2 –	record containing code/data and 3–byte address at which the code/data is to reside.
S8 –	termination record.

5 Packed Motorola S Records

The general format of a record, shown here with spaces separating each field, is as follows:

ss ll aaaa[*aa*[*aa*]] *dd...dd cc*

Where:

ss	is the S–record type.
ll	is the record length, which includes the number of bytes in the address, code/data and checksum fields.
aaaa[*aa*[*aa*]]	is the 2–, 3– or 4–byte address at which the first byte of code/data in the record is to be placed.
dd...dd	is the code/data bytes.
cc	is the checksum.

The following is a list of the possible S–record types for the *ss* field:

S0 –	header record for each block.
S1 –	record containing code/data and 2–byte address at which the code/data is to reside.
S2 –	record containing code/data and 3–byte address at which the code/data is to reside.
S3 –	record containing code/data and 4–byte address at which the code/data is to reside.
S7 –	termination record that includes a 4–byte start address.
S8 –	termination record that includes a 3–byte start address.
S9 –	termination record that includes a 2–byte start address; also if there is no defined start address (two bytes of zero).

6 S37 Motorola S Records

The general format of a record, shown here with spaces separating each field, is:

ss ll aaaaaaaa dd..dd cc

Where:

ss	is the S–record type.
ll	is the record length, which includes the number of bytes in the address, code/data, and checksum fields.
aaaaaaaa	is the 4–byte at which the first byte of code/data in the record, code/data, and checksum fields.
dd...dd	is the code/data bytes.
cc	is the checksum.

The following are possible S–record types for the *ss* field:

S3 –	record containing code/data and 4–byte address at which the code/data is to reside.
S7 –	termination record.

Note: This format does not provide an S0 header record.

7 Tektronix Format (Tekhex)

The general format of a record, shown here with spaces separating each field, is:

/ aaaa ll ss dd...dd cc

here:

/	is the keyword used to signal the start of a record.
aaaa	is the 2–byte address at which the first byte of code/data in the record is to be placed. Successive data bytes are stored in the following memory locations.
ll	is the number of code/data bytes in the record. A count of zero indicates end–of–file.
ss	represents the sum of the preceding six digits. (a+a+a+a+l+l).
dd...dd	is the code/data bytes.
cc	is the checksum.

8 Extended Tekhex Format

The formatter can produce extended Tektronix Hexadecimal Format (Extended Tekhex). Symbolic information is produced for global

symbols when the the formatter debugging switch, -d, is used. To conform with legal Extended Tekhex conventions, symbol names are modified as follows before being emitted:

- Leading underscore ('_') characters are removed.
- Illegal characters ('@' and '*') are replaced with '$'.
- Leading dollar sign ('$') characters are moved to the end of the name.
- Symbol names longer than 16 characters are truncated.

The general format of a record, shown here with spaces separating each field, is:

% ll t cc aa(aaaaaaaaaaaaaaaa) dd...dd

Where:

%	is the keyword used to signal the start of a record.
ll	is the number of digits in the record (not including the leading % or end–of–line).
t	indicates the record type.
cc	checksum
aa	may be from 2 to 17 hex or ASCII digits. The first digit is always a hex which indicates how many digits are to follow. The meaning of theaa field depends on the type of record. (See below.) data: load address of object code symbol:name of thesection that contains the symbols defined in this block termination: transfer address, or the address where the program must begin.
dd	Meaning and length depend on record type. (See below.) data Each dd represents a byte of object code/ data. symbol 5 to 35 hex digits of Section Definition and 5 to 35 ASCII and hex digits for each Symbol Definition. termination No characters in this field.

Record Types:

The following are possible record types for the *t* field:

6 =	data block
3 =	symbol block
8 =	termination block

8.1 Section Definition Field

The general format of a section definition is as follows, with a space separating the fields. (This is an expansion of *dd* above.)

0 *AA LL*

Where:

0	is the keyword used to identify the start of a section definition.
AA	is 2 to 17 hex digits which represent the starting address of the section. The first digit indicates how many digits will follow.
LL	is 2 to 17 hex digits which represent the length of a section. The first digit indicates how many digits will follow.

8.2 Symbol Definition Field

The general format of a symbol definition field is as follows:

X SS VV

Where:

X	Indicates the type of value that the symbol represents. For our purposes, this hex digit is always a '1', meaning a global address.
SS	is 2 to 17 digits which represent the name of the symbol. The first digit is a hex digit which indicates how many ASCII digits will follow.
VV	is 2 to 17 hex digits which represent the address of the symbol. The first digit indicates how many digits will follow.

9 Binary Tektronix Format

Binary Tekhex format is a binary format. A full description of this format is beyond the scope of this appendix. For details, please refer to the Tektronix binary object format specification.

The following Binary Tekhex records are produced when the formatter debugging switch, `-d`, is *not* used:

- `LAS_MODULE_DEFINITION_BLOCK`
- `MODULE_COMMENT_INFO_BLOCK`
- `MICROPROCESSOR_DEPENDENT_BLOCK`
- `SECTION_EXPORT_BLOCK` : `LAS_SECTION_DEFINITION_RECORD` only
- `LAS_TEXT_BLOCK`
- `LAS_END_BLOCK`

In addition, global symbol information is available with the following record when the formatter debugging switch, `-d`, *is* invoked:

`LAS_MODULE_SYMBOL_TABLE_BLOCK` :
`LAS_GLOBAL_LABEL_RECORD` only

Symbol names longer than 16 characters are truncated.

10 HP64000 Format

HP64000 format is a binary format. A full description of this format is beyond the scope of this appendix. For details, please refer to the Hewlett Packard document *64000–UX Hosted Development System File Formats* (HP 64880–90903).

10.1 Using the HP64000 Format

The formatter can produce all the files necessary for doing:

For PC hosts: Emulation analysis on a HP64700 emulator using the Emulator Interface.

For Unix hosts: Emulation and state analysis on a Hewlett Packard HP 64000–UX system. Output is compatible with the following systems:

- Hewlett Packard Emulation Bus Analyzer (HP 64302A) and the Hewlett Packard State/Software Analyzer (HP 64620S).
- Hewlett Packard 32–bit Emulation Bus Analyzer (HP 64416A/B Real–Time Emulator).

All global, local static, and source line symbols are available for all hosts.

For Unix hosts, the files that are generated by the formatter utility are appropriate for downloading from a UNIX development environment to the HP 64000–UX with:

- The Hewlett Packard downloading program called `get64` (HP 64887S Network Transfer Utility), which runs on the HP 9000 Series 300 running HP–UX.

  The `get64` program performs a translation of these downloadable files from a UNIX–specific format to an HP 64000–UX specific format during downloading. If the formatter is hosted on an HP 64000–UX it generates these files in a format which is already HP 64000–UX specific.

10.2 Files Needed

The following file types are required for emulation analysis (and state analysis for Unix hosts) on the HP64700 for PC hosts and HP64000–UX for Unix hosts:

- **Absolute file (.X).** The absolute file contains absolute data that will be loaded into memory, and information about the target processor that will be used. One absolute file is needed per program.
- **Linking locator symbol file (.L).** The linking locator symbol file contains global symbol definitions, lists source modules, and describes the location of the program code that was generated from each source module. One linker symbol file is required for each program.
- **Assembler symbol files (.A).** Assembler symbol files contain local static symbol definitions and information for displaying line numbers during emulation analysis (and state analysis for Unix hosts). One assembler symbol file is required for each source module for which source lines and/or local static symbols will be displayed.
- **Source files.** The source files used to generate the executable module are required for displaying source line information during emulation and state analysis.

All of the necessary files are created by the formatter. From each absolutely located object file produced by the linking locator (.ab file), the formatter produces one absolute file, one linking locator symbol file, and optionally, assembler symbol files. For an overview of the way that these files are used on the HP 64000–UX for Unix hosts and a detailed description of the file formats for PC and Unix hosts, see the Hewlett Packard document, *64000–UX Hosted Development System File Formats* (HP 64880–90903).

10.3 Generating Files for Use with the 64700 for PC Hosts or for Downloading on UNIX for Unix Hosts

To produce files for use with the HP64700 for PC hosts or the HP64000–UX for Unix hosts, the tools are run as usual, but with the following additions:

- If source line information and/or local static symbol information is to be made available for a particular source module (i.e., if an assembler symbol file is to be produced for that module), then that source **must** be compiled or assembled with the `-d` (debugging) switch.
- Choose the HP64000 format by running the formatter with the `-f hp` switch.
- If source line information and/or local static symbol information is required for any module, use the `-d` formatter switch.

Generally, source line information is needed for only part of the input source modules. For example, displaying source lines for assembly language modules will not always be useful, since it is possible to disassemble code in the state and emulation analyzers. Also, run–time libraries are usually not assembled with debugging information. Any combination of object language files compiled or assembled with and without debugging information may be linked together. Assembler symbol files will only be produced for those that have been compiled or assembled with debugging information.

Note: At formatting time, if an assembler symbol file is created, it is placed in the same directory as its corresponding source file (not necessarily the same directory in which the formatter is invoked). Therefore, source directories that are used for this purpose must be writable.

10.4 Formatter Examples for PC and Unix Hosts

In the following examples, it is assumed that two source files, xtest.c and ytest.c, have been compiled (using the `-d` switch), linked and located, producing an absolute file called xtest.ab.

Example

```
form xtest.ab -f hp
```

- For PC hosts only, produces an absolute file named `xtest.X`, and a linker symbol file named `xtest.L`.
- For PC and Unix hosts, neither source line information nor local static symbol information is included.

Example

```
form xtest.ab -f hp -d
```

- Produces an absolute file, `xtest.X`, a linker symbol file, `xtest.L`, and assembler symbol files, `xtest.A` and `ytest.A`, containing source line information and local static symbol information for the corresponding source files.

10.5 Using get64 on Unix Hosts

The files that are generated by the formatter are in a format appropriate for downloading to the HP 64000–UX with the Hewlett Packard program `get64`, which runs on the HP 9000 Series 300 running HP–UX. For a description of the options to `get64`, see the HP manual *Network Transfer Utility for the HP 64000–UX Microprocessor Development Environment* (HP 64887–90901).

The absolute file, the linker symbol file, the assembler symbol files, and the source files may all be downloaded to the HP 64000–UX at the same time. The name of the linker symbol file (with an absolute directory path) is specified as an argument to `get64`. `get64` derives the name of the absolute file from the name of the linker symbol file by replacing the ".L" extension with ".X". The names of the source files, with absolute directory paths, are included in the linker symbol file, and the assembler symbol files, if created, are found by `get64` in the same directory as their corresponding source files (the name of the assembly symbol file is inferred from the source file name).

10.6 InterTools and get64 on Unix Hosts

When compiling (or assembling) files with the InterTools products, it is not necessary to specify absolute directory paths for the names of the sources. Using the names with which the compiler (or assembler) was invoked, the formatter will create file names with full directory information and insert them in the linker symbol file. Therefore, *the compiler, assembler and formatter must all be executed in the same directory.*

If the source is specified with a directory path relative to the current working directory, the absolute path name that the formatter builds for the file and places in the linker symbol file will not necessarily be the shortest name possible. For example, suppose the InterTools compiler is invoked within the directory `/users/itools/test1`, with the name of the source specified as `../test2/test.c`. At formatting time, the formatter will create in the linker symbol file an absolute directory path name `/users/itools/test1/../test2/test.c`, and **not** `/users/itools/test2/test.c`.

The following points are important to keep in mind:

- when composing a file of mapping patterns to be used with the `-m` option of `get64`. If you are unsure of the absolute path file names included in the binary linker symbol file, run the UNIX program `strings` on the linker symbol file to determine the full names of the source files.

- for viewing local symbol information when using the HP 64416A/B Real Time Emulator. In order to display local symbols for a file with `../` in its directory path, you must precede each occurrence of `../` within the directory path with a backslash `\`.

- if the source file has no extension in its name (i.e., `../test2/test`, as opposed to `../test2/test.c`). The HP 64416A/B Real Time Emulator, using the source file names within the linker symbol file, is unable to correctly infer the name of the assembler symbol file corresponding to a source file with no extension and with a `../` in its directory path (even though the assembler symbol file has been created in the proper directory). If this occurs, local static symbol information and source line referencing information will not be available within the emulator for that source file.

11 Common Object File Format (COFF)

COFF is a binary format. A full description of this format is beyond the scope of this appendix. There are several implementations of the COFF standard, which differ largely in regard to which fields of the COFF records are filled in and which are null padded. Unless otherwise noted below, the InterTools formatter utility fills in fields according to the *System V/68 Release 3 Programmer's Guide* (Motorola, Inc., 1987) and *Understanding and Using COFF* by Gintaras R. Gircys (O'Reilly & Associates, Inc., 1988).

COFF output from the formatter has been tested with the ATRON 68000/010/020 series emulators. Line numbers and all global and local symbols are available.

The following table shows the basic structure and contents of a COFF file (from figure 1–2 in *Understanding and Using COFF*).

11.1 File Header

The only magic number (`f_magic`) used currently in the file header is octal 520. This is normally associated with an MC68000 target processor. The formatter utility does process other target `.ab` files, but the resulting COFF file always gets this magic number.

Note that for the Intel family of processors, the byte ordering in the binary COFF file is done in the following manner: the least significant byte gets the lowest address. This is the reverse of the ordering used by the MC68000. The `-br` formatter switch allows both types of byte ordering for the supported processors, to better allow for cross development.

A COFF File	Contents
File Header	General information such as file timestamp and magic number
Optional Header	Run time information
Section Header 1 . . Section Header	Descriptions of section characteristics
Section 1 contents . . Section *n* contents	Actual contents of sections
Section 1 relocation info . . Section *n* relocation info	Information used by the linker to create run time executable
Section 1 line number info . . Section Header	Debug information
Symbol Table	Information used by the debugger and linker
String Table	Very long symbolic names

11.2 Option Header

The `vstamp`, `text_start`, and `data_start` fields are not used (null padded). The magic number is always set to octal 520. This means that text and data segments are aligned within the binary file, so that the file can be paged directly.

11.3 Relocation Information

This information is not emitted by InterTools, since the `llink` utility will have already been used for relocation.

11.4 Section Headers

The relocation information related fields `s_relptr` and `s_nreloc` are always zero. The section names `idata` and `udata` are equivalent to `.data` and `.bss` respectively.

The only types of sections emitted are: `STYP_TEXT`, `STYP_DATA`, `STYP_BSS`.

The names of the sections are truncated to the eight character limit.

11.5 Line Number Information

Line number entries are absent when files are compiled without the symbolic debug switch.

11.6 Symbol Table Entries

Symbol table entries are absent when files are compiled without the symbolic debug switch. The following items list the different implementation techniques used for Symbol Table entries:

- External symbols receive an extra prepended '_' (underscore) character. The `-c` formatter switch suppresses the addition of an extra underscore.
- There are several "special symbols" that are not supported. They are: `.text`, `.bss`, `.data`, `.target`, `.bb`, `.eb`, `etext`, `edata` and `end`.
- The auxiliary information for `DT_FCN` function entries contains only the `x_lnnoptr` line number pointer and the `x_fsize` function size fields; other fields are not used.
- The `.bf` and `.ef` auxiliary directives do not contain the "line number" or "index of next entry" information and are left null padded.
- Tag type entries are not chained.

11.7 COFF1 Format

COFF1 format is almost identical to the COFF format described earlier in this appendix.

It differs only in its treatment of line number symbols. COFF1 format starts its line numbers from 1. As a result, each line number symbol directly corresponds to an actual program line number.

Note that this is not true for the InterTools COFF format, which was developed for use with the ATRON emulator. The ATRON emulator requires COFF line number symbols which have been specially processed. Therefore, COFF line number symbols do not directly correspond to the program line numbers.

APPENDIX E

Compiler/Assembler Driver

This appendix discusses the driver that invokes compiler and assembler executables.

1 For PC and Unix Hosts:

The compiler and assembler are invoked by a `driver` program. This program is responsible for reinvoking the compiler and/or assembler repeatedly if there are multiple source files, and for invoking the various executables which constitute the compiler. This `driver` program has switches of its own, which are generally of interest only when debugging the compiler and/or the assembler itself. They are:

`-ke`	For PC hosts only, keep the intermediate files between compiler phases. For Unix hosts only, execute the phases of the compiler sequentially and keep the intermediate files.
`-se`	For Unix hosts only, this switch is like `-ke`, but deletes the intermediate files if compilation is successful.
`-v`	Verbose mode. Identifies executables as they are invoked. This helps determine which phase was executing if the compiler aborts. For technical support purposes.
`-V`	Very verbose mode. Reports date, time, and status/result.

For PC hosts only, the driver by default queries the `I2EXE` environment variable to determine the directory containing executable files. If the `I2EXE` variable is not defined, the driver invokes executable files which lie in the same directory as the driver executable.

For Unix hosts only, the driver by default invokes executable files which lie in the same directory as the driver executable.

The following switches direct the driver to invoke other executables.

`-as` *file*	Specifies a different assembler executable.
`-be` *file*	Specifies a different compiler back end executable.
`-fe` *file*	Specifies a different compiler front end executable.
`-gs` *file*	Specifies a different global symbol mapper utility.
`-in` *file*	Specifies a different interleave utility.
`-me` *file*	Specifies a different merge utility.
`-op` *file*	Specifies a different compiler optimizer.
`-xd` *directory*	Specifies a different executable directory.
`-xr` *file*	Specifies a different cross–reference utility.
`-1`	Invokes the assembler once, passing all source files. When using other switches with `-1` (one), `-1` must appear first in the invocation.

For PC hosts only, the driver program uses the `I2ARGV0` environment variable internally. You should not use this variable for any other purpose.

APPENDIX F

Supplemental Reading

This chapter contains a list of other sources of information that may be of interest to programmers using the InterTools system.

1 Bibliography

For additional information about programming for embedded systems, you may want to consult one or more of the following:

7. *American National Standard for Information Systems – Programming Language C*
   ANSI/ISO 9899-1990, 1990
8. *The C Programming Language*
   Kernighan, Brian W. and Ritchie, D.M.
   Prentice–Hall, Inc.
   Englewood Cliffs, New Jersey
   (ISBN # 0–13–110163–3)
9. *M68300 Family CPU32 Central Processor Unit Reference Manual,* 1st Ed.
   Motorola Semiconductor Products, Inc., 1990
   Phoenix, Arizona
10. *MC68000 Systems Resident Structured Assembler Reference Manual,* 8th Ed.
    Motorola Semiconductor Products, Inc., 1984
    Phoenix, Arizona
11. *MC68020 32–Bit Microprocessor User's Manual,* 2nd Ed.
    Prentice–Hall, Inc.
    Englewood Cliffs, New Jersey

12. *MC68030 Enhanced 32–bit Microprocessor User's Manual*, 2nd Ed.
Prentice–Hall, Inc., 1989
Englewood Cliffs, New Jersey

13. *MC68040 32–bit Microprocessor User's Manual*, 1st Ed.
Motorola Semiconductor Products, Inc., 1989
Phoenix, Arizona

14. *MC68851 Paged Memory Management Unit User's Manual*, 1st Ed.
Prentice–Hall, Inc., 1989
Englewood Cliffs, New Jersey

15. *MC68881/MC68882 Floating–Point Coprocessor User's Manual*, 1st Ed.
Prentice Hall, Inc., 1987
Englewood Cliffs, New Jersey

16. *MC68332 User's Manual*
Motorola Semiconductor Products, Inc., 1990
Phoenix, Arizona

17. *MC68302 Integrated Multiprotocol Processor User's Manual*
Motorola Semiconductor Products, Inc., 1991
Phoenix, Arizona

18. *MC68340 Integrated Processor with DMA User's Manual*
Motorola Semiconductor Products, Inc., 1992
Phoenix, Arizona

19. *MC68360 Quad Integrated Communications Controller User's Manual*
Motorola Semiconductor Products, Inc., 1993
Phoenix, Arizona

20. *C: A Reference Manual*
Harbison, Samuel P. and Steele, Guy L. Jr., C
Prentice–Hall, Inc., 1987
Englewood Cliffs, New Jersey
(ISBN # 0–13–109810–1)

Index

Symbols

#elif directive, A–27–A–40

#error directive, A–27–A–40

#pragma directive, A–27–A–40

_GPL pseudo–function, A–33–A–40

_IH keyword, A–31, A–32, A–34–A–40

_LONGINT variable, 2–23

_SPL pseudo–function, A–33

_SWI keyword, A–36–A–40

_TRAP function, A–34–A–40

Numbers

16–bit subscript, 2–79

64K limit, 7–17, C–4

A

A0 register, C–5

A5, libraries that do not use A5, 7–21

A5 register, 2–47–2–48, 2–53, 7–14

A5 relative addressing, 7–17, 7–21, C–4

A5–relative addressing, 2–47–2–48, 2–53

A6 register, 2–54, C–3

absolute
  module, 4–7, 5–6, 5–7
  segment, 4–27

address
  range, 4–36–4–52
  reference, 4–26

address exception, 2–25–2–26

address spaces, multiple, 3–8, 4–43

addressing modes, 2–48, 2–57, 2–58, 2–59, 2–60, 2–80–2–86, 7–44, 7–61

AFTER address, 4–36, 4–40

alignment, 2–24–2–26, 2–44, 3–17

ANSI C, 2–6, 2–38–2–39, 2–42, 2–56, 2–62, 2–63, 2–68, A–16–A–40

ANSI function prototypes, 2–68

array subscript, 2–79, 2–80

ASCII hex formats, D–1

assembler, 3–1–3–26
  driver, E–1–E–2
  options, listing, 3–7–3–26
  usage, 3–6–3–26

assembly
  conventions, 7–9

in–line, A–3–A–40
routine call, 7–10
assembly listing, 2–13
automatic register variable assignment, 2–73–2–86

B

backwards compatibility, 2–21, 2–39
BEFORE address, 4–36
bias, 5–16
bibliography, F–1–F–4
binary formats, 5–13, D–12–D–22
Binary Tekhex format, 5–8, 5–9, 5–13, D–11–D–22
bitfield storage layout, 2–21
branch tables, 2–78–2–86
built–in datatypes, 2–21
burning interleaved PROMs, 5–17
byte slicing, 5–17, 5–19

C

C and assembly, 7–14
linking, 7–9–7–16
C interrupt handlers, A–31–A–40
C language specifications, A–1–A–40
C++ support, 4–29
calling, sequence, C–6–C–10
cataloging object modules, 6–7
checksums, D–1
class, 4–24–4–32, 6–30, 7–19, B–6, B–7
code
generation, options, 2–44–2–86
hoisting, 2–34, 2–76–2–86
segment, B–5–B–10
size, 2–74
symbols, B–2–B–10
COFF, 5–9, 5–13, 5–19, D–19–D–22
COFF1, 5–13
combinability, 4–21
combining segments, 4–21
common
segment, 4–21
subexpression elimination, 2–74–2–86
compiler, 2–1–2–28
driver, E–1–E–2
error messages, 2–80–2–86, A–20
input, 2–2
invocation, 2–1
library organization, 4–27–4–32
options, 2–2–2–28
output, 2–2
runtime conventions, C–1–C–10
usage, 2–10–2–28
concat segment, 4–21
conditional assembly, 3–8, 3–9
const, A–23
const type qualifier, 7–19, 7–20, A–23–A–40
constant data, 7–18

constants in ROM, 7–18

conventions
  compiler, A–3
  compiler naming, 7–10–7–16
  compiler runtime, C–1–C–10
  documentation, 1–4–1–8

cross–reference listing, 2–14, 2–15, 2–16

CrossView Pro, debugger, 2–11, 7–6

CSE, 2–74

customer support, 2–81

D

D0 register, C–5

data
  allocation rules, C–1
  global, B–3–B–10
  local static, B–3, B–4–B–10
  segment, B–5–B–10
  separate, B–6–B–10
  stack, B–3, B–4–B–10
  symbols, B–3
  uninitialized, B–5, B–6, B–7

datatype
  length options, 2–22
  options, 2–19–2–28
  void, A–32

date/time stamp, 6–9, 6–11

debugging
  information, 2–62, 3–18
    symbolic, 3–18, 4–15
  with optimizer, 2–71

DECLARE command, 4–38

default
  format, 5–8
  placement, 4–26
  separate
    class, 2–26, 7–19
    segment, 7–19

defined operator, A–29–A–40

directives, ANSI C, A–27–A–40

documentation, 1–1–1–8
  conventions, 1–4–1–8

download
  file, 5–23
  format, 5–8, D–1–D–22

downloading, 7–5–7–8

driver, E–1–E–2

E

empty segments, 6–18

emulators, 7–5–7–8

entry/exit optimization, 2–78–2–86

enum type, 2–19

error messages
  compiler, 2–80–2–86, A–20
  formatter, 5–27–5–30
  global symbol mapper, 6–20
  librarian, 6–11
  linking locator, 4–49–4–52
  object size list utility, 6–32
  symbol list utility, 6–27

exception, A–31
  handler, A–32
  vector, A–32

excluding from download
named classes, 5–24
named segments, 5–23

Extended Motorola format, 5–14, D–4–D–22

Extended Tekhex format, 5–8, 5–9, 5–13, 5–14, D–7–D–22

extensions, preprocessor, A–3–A–40

external, reference, 4–17, 4–33

F

floating point compatibility, 2–39

formats, object module, D–1–D–22

formatter, 5–1–5–30
error messages, 5–27–5–30
input, 5–1
invocation, 5–1
options, 5–2–5–30
output, 5–1, D–1–D–22
usage, 5–6–5–30

fullword alignment, 2–25–2–26, 3–17

function
declaration syntax, A–17
definition header, A–17
header, A–16
prototypes, A–16–A–40
creating, A–16–A–40
return temporary, C–6
return values, 2–39

function prototype, 2–69

G

get64 program, D–17

global
consistency check, 6–9, 6–11
data, 7–14, 7–17, B–3–B–10, C–3, C–4
sharing, 7–14–7–16
function, B–2, B–3, B–5
replace operation, 6–10
symbol, 4–15, 6–5, 6–11, 6–12, 6–15, B–1, B–2, B–3
listing, 3–4, 3–8
variable, 4–19

global symbol mapper, 6–13–6–20, B–2
input, 6–13
invocation, 6–13
listing, 6–15
options, 6–14–6–20
listing, 6–17
sorting, 6–19–6–20
output, 6–13
usage, 6–15–6–20

GPL. *See* _GPL pseudo–function

group, 4–22–4–32

gsmap. *See* global symbol mapper

H

halfword alignment, 3–17

handler, A–32

hardware floating point, 2–39, 2–41, 2–42, 4–28

hardware floats option, 2–39

Hewlett–Packard format. *See* HP64000 format

hex file, 3–18, 5–24, 7–6
HP64000 format, 5–8, 5–9, 5–13, D–12–D–22

I

idata, 2–61, 4–5, 4–22, 7–17, 7–18, B–5, B–9
IH. *See* _IH keyword
in–line, assembly, A–3–A–40
in–line assembly, 2–13, 2–62, A–3
include
  directive, 2–17
  options, compiler, 2–16–2–28
initial
  stack frame, C–9–C–10
  values, 4–5, 4–19
initialization, segment, 4–6
initialized variables, 4–5, 7–17, 7–20
input
  assembler, 3–2
  compiler, 2–2
  formatter, 5–1
  global symbol mapper, 6–13
  librarian, 6–3
  linking locator, 4–1
  object size list utility, 6–29
  symbol list utility, 6–21
instruction, for returns, A–34
Intel
  ASCII hex format, 5–13
  formats, D–2–D–22
internal symbol, B–1, B–2, B–3
interrupt
  handlers, A–32
    C, A–31–A–40
  priority level, A–33
interrupt handlers, 2–11, 2–72
invocation
  compiler, 2–1
  formatter, 5–1
  global symbol mapper, 6–13
  librarian, 6–3
  linking locator, 4–1
  object size list utility, 6–29
  symbol list utility, 6–21

L

ldata, 4–23, B–5
libr, 6–3
librarian, 6–3–6–12
  input, 6–3
  invocation, 6–3
  options, 6–3–6–12
    command, 6–7–6–12
  output, 6–3
  usage, 6–5–6–12
library
  index
    file, 4–2, 4–4, 4–9, 6–5, 6–6, 6–10
    header, 6–5, 6–11
  listing output, 6–11
  search, 4–10, 4–33–4–52, 6–9
lifetime
  analysis, 2–36, 2–72
  overlap, 2–72
LINK instruction, 2–35
linkage conventions, 2–39, 2–40

linking, 4–4–4–18
  options, 4–9–4–18

linking locator, 4–1–4–18
  error messages, 4–49–4–52
  input, 4–1
  invocation, 4–1
  options, 4–2–4–18
  output, 4–1
  usage, 4–4–4–18

listing
  object size, 6–29–6–32
  symbol table, 6–24–6–28

listing options
  assembler, 3–7–3–26
  compiler, 2–12–2–28
  global symbol mapper, 6–17–6–20
  librarian, 6–11–6–12

llink. *See* linking locator

loader program, 5–6, 7–38

local
  functions, B–2
  information, 4–15
  stack data, B–3, B–4–B–10

LOCATE command, 4–19, 4–24, 4–26, 4–40

locating, 4–7
  segments, 4–23

locator, commands, 4–10, 4–34–4–52

long integer, 2–23, 4–29

loop rotation, 2–77–2–86

M

M68020, 2–24–2–26

M68030, 2–24–2–26

M68040, 2–24–2–26

M68EC020, 2–24–2–26

M68EC030, 2–24–2–26

M68EC040, 2–24–2–26

macro, definition, A–25

macros, ANSI predefined, A–26–A–40

main routine, C–9

mathematical functions, 2–42

maximum address, 4–42

MC68000, 2–1, 3–2
  compatibility mode, 2–39
  instruction set, 2–11, 3–6

MC68010 instruction set, 3–6

MC68020
  addressing mode, 2–80
  instruction set, 2–11, 3–6

MC68030 instruction set, 3–6

MC68040, 2–44

MC68302, 2–11, 3–3, 3–4, 3–6, 3–15, 3–19

MC68332 instruction set, 2–11, 3–6

MC68340, 2–11, 3–4, 3–6, 3–19

MC68881, instructions, 4–28

memory
  limitation, 4–42
  management, 4–43

MEMORY command, 4–27, 4–42

memory mapped I/O, 2–11, 2–72, 4–20
memory space, 4–19
messages
  global symbol mapper, 6–20
  librarian, 6–11
  object size list utility, 6–32
  symbol list utility, 6–27–6–28
MICROCASE SoftAnalyst, 5–11
missing, routines, 4–39
Motorola S format, 5–13, D–3–D–22
multiplication optimization, 2–79–2–86

N

name
  conflict, 4–15
  list, 4–37, 4–40
negative offsets, C–3
nested include, 2–17
nesting limit, 2–17
no–A5 library, 2–53
no–alias option, A–28
no–floats library, 4–29
non–volatile option, A–28
null class, 4–25
NWIS ASCII format, 5–11, 5–15

O

object
  module, 2–10, 6–7
    cataloging, 6–7
  text window, 5–18
object size list utility, 6–29–6–32
  input, 6–29
  invocation, 6–29
  listing, 6–31
  messages, 6–32
  options, 6–29–6–32
  output, 6–29
  usage, 6–30–6–32
offsets
  negative, C–3
  positive, C–3
olsize. *See* object size list utility
operators, ANSI C, A–29–A–40
optimizer, 2–11, 2–73–2–86
  and debugging, 2–4, 2–11, 2–32
  options, 2–29–2–86
  special instructions, 2–80–2–86
  subexpression, 2–29
  suppression, 2–71
  usage, 2–71–2–86
option sep_off, 7–19
option sep_on, 2–26–2–28, 7–19, 7–20
option separate, 2–26, 5–26, 7–17–7–20, B–5, B–6–B–10, C–3
options
  assembler, 3–7–3–26
  compiler, 2–2–2–28
  formatter, 5–2–5–30
  global symbol mapper, 6–14–6–20
    listing, 6–17–6–20

sorting, 6–19
librarian, 6–3–6–12
command, 6–7
listing, 6–11–6–12
linking locator, 4–8–4–18
listing, librarian, 6–11–6–12
object size list utility, 6–29–6–32
symbol list utility, 6–22–6–28
output
assembler, 3–2
compiler, 2–2
formatter, D–1–D–22
global symbol mapper, 6–13
librarian, 6–3
linking locator, 4–1
object size list utility, 6–29
symbol list utility, 6–21, 6–23
overlapping segments, 4–27

P

Packed Motorola format, 5–13
padding segments, 4–10, 4–47
pagination, 3–9
suppression, 6–17
parameter passing, C–5–C–10
PC–relative addressing, 7–38–7–52
PC–relative code, 2–58
peripherals, on–board, 7–63
portable libraries, 6–6
position–independent code, 2–57, 2–58, 2–59–2–62, 3–16, 7–37–7–52
position–independent data, 7–41
positive offsets, C–3
pragma sep_off, 7–19
pragma sep_on, 2–26–2–28, 7–19, 7–20
pragma separate, 2–26, 5–26, 7–17–7–20, B–5, B–6–B–10, C–3
pre–INCLUDE'd files, 3–15
prelink, 4–5
preprocessor, 2–64, 2–65
additions, ANSI C, A–26–A–40
extensions, A–3–A–40
option directives, 7–18–7–20
preserved registers, 2–49, C–4
procedure
blocks, C–3
call, 2–49
prologue, 2–35, C–6–C–10
program image, 7–6
prologue, C–6–C–10
PROM
burners, 7–6, 7–7
options, 5–16–5–30
programming, 7–5, 7–6–7–8
prototype. *See* function prototype
pseudo–assembly, listing, 2–10

R

RAM, 4–5, 7–8
rcopy, 4–6
read–only variables, A–23

register
allocation, 2–36, 2–72, 2–73
preserved, C–5–C–10
return values, C–5–C–10
usage, C–4–C–10
variable, 2–50, 2–73, 2–75

relocatable segment, 4–26

relocation, 4–26–4–32

RESERVE command, 4–26, 4–45

return values, 7–13

ROM, 4–5, 7–6, 7–8, 7–18
processor, 4–4
variables in, A–23

rompOutSeg, 4–5

RS–232 connection, 7–6

RTE instruction, A–32, A–34

RTS instruction, A–34

runtime conventions, 2–49

runtime model, 2–47, 2–49

runtime library, 2–20, 2–22, 2–41, 2–47, 2–53, 2–54, 4–18

runtime conventions, C–1–C–10

runtime library, 7–21
organization, 4–27–4–32

S

S37 Motorola S records, D–6–D–22

sdata, 2–61, B–5

segment, B–1, B–5–B–10
absolute, 4–27
code, B–5–B–10
data, B–5
length, 4–19
name, 4–19, 7–19, B–2
overlapping, 4–27
padding, 4–47

segmentation model, C–2–C–10

separate
data, 2–26–2–28, 7–20, B–6–B–10
default class, 2–26, 7–19
default segment, 7–19

signed bitfield, A–37

signed/unsigned attribute, 2–22

simulators, 7–5–7–8

software
floating point, 2–39, 4–28
interrupt keyword, A–36–A–40

source, listing, 2–16

SPL. *See* _SPL pseudo–function

stack, 2–54, 2–55
data, B–3, B–4–B–10
fixup, 2–55
frame, 2–35, 2–54
initial, C–9–C–10
layout, C–1
traceback, 2–35

START command, 4–48

static
data items, 7–17
functions, B–5
keyword, B–2
variable, B–9

storage allocation, 2–21, 2–24–2–26, C–1–C–10

strength reduction, 2–37, 2–76–2–86
string constants, B–4–B–10
stringization, A–25–A–40
structure, fields, 2–65
structure, fields, 2–24–2–26
structured assembly, 3–9
subroutine, linkage, C–5–C–10
subscript optimization, 2–79–2–86
supplemental reading, F–1–F–4
support, customer, 2–81
SWI. *See* _SWI keyword
symbol
  formats, 5–6, 5–8, 5–12
  global, B–1, B–3
  information, 2–62, 3–18, 4–15, 5–8
  internal, B–1, B–3
  naming, B–8–B–10
symbol list utility, 3–18, 6–21–6–28, B–2
  input, 6–21
  invocation, 6–21
  listing, 6–27
  messages, 6–27–6–28
  options, 6–22–6–28
  output, 6–21, 6–23
  usage, 6–22–6–28
symlist. *See* symbol list utility
system, include directory, 2–16

T

target path
  computation, 2–75–2–86
  optimization, 2–75
Tekhex format, 5–14, D–7–D–22
token pasting operator, A–30–A–40
TRAP, A–31
trigraphs, A–22–A–40
type
  char, 2–19
  double, 2–39
  float, 2–39
  int, 2–19, 4–29
  qualifier, const, A–23–A–40
  short, 2–19
type qualifier, A–28

U

udata, 4–22, 5–23, 7–17, 7–18, B–5, B–9
unassembled source, 3–9
uncombinable segment, 4–21
undefined symbol, 2–69, 4–4
uninitialized
  bytes, 4–19
  data, B–5, B–6, B–7
  storage, 4–19
  variables, 7–17, 7–20
updating library, 6–5, 6–6, 6–9, 6–10
uppercase identifiers, 3–18
user include
  directory, 2–16

file, 2–16

V

VBR register, A–31
vector exception, A–32
void
  datatype, A–32
  keyword, A–18
  pointers, A–23–A–40
volatile, 2–37–2–39, 2–72, A–28
  keyword, A–32

W

warning
  messages, 2–70, 3–19
  severities, 2–70, 3–19

X

XDB
  debugger, 2–29–2–44
  formatter options, 5–20–5–30
XDB debugger, 2–54

Z

Z80SBC format, 5–14
ZAX format, 5–11, 5–16

RELEASE NOTE

INDICATOR	:	Customer Information Software
INDICATOR NR.	:	CIS9847
CONCERNS	:	TK0122–00 68xxx InterTools Release 9.0
ISSUE DATE	:	DEC '97
SUPERSEDES	:	—
TO BE FILED IN	:	68xxx InterTools User's Guide

SUMMARY

A new release of the 68xxx InterTools has been made: Release 9.0.

The main reasons for this update are:

- C++ support
- CrossView Pro debugger compatability

7.5 BUILDING LIBRARIES THAT DO NOT USE A5

Sometimes users may wish to eliminate the A5-relative addressing that is used by default for global data. This can be accomplished with command line switches which have the effect of making all global data "separate", that is, directly addressed. However, A5-relative addressing may still remain, because the runtime library has some private global data of its own which is addressed via A5. This application note tells you how you can build a library that does not use any A5-relative addressing.

The runtime library contains both C and assembly source files. The C modules must be recompiled using switches that make all data separate, plus any other switches, such as -L, that are needed for your particular application. Subsequent links will run faster if you pre-link each object module by itself (to resolve internal references), but this is not essential. Alternate versions of the assembly language modules that reference data via A5 are supplied with the library distribution, so no assemblies need be done. After the C modules are recompiled, the library index files must be updated to include the alternate assembly language modules and the recompiled C modules.

There are several different libraries supplied with the distribution. There are libraries for different targets, like MC68000 or MC68020. There may be libraries for hardware and software floating point. Within each of these are libraries for use with -L (long integer) and libraries for use without -L (the default). There are libraries that do not support floating point (the "no-floats" libraries), and libraries that do support floating point (the default). Of course, you need only rebuild the libraries you intend to use. The examples that follow show you how to rebuild all the libraries for the MC68000 software floating point target. The procedure for other targets is similar, except that you use a different compiler name, for the specific target (e.g., 68020, 68040, etc.). In addition, the command line switch for hardware floating point (-h) must be used for libraries which rely on floating point hardware.

At the end of this chapter there are six tables labelled "Table 7-1" through "Table 7-6". Each table represents a subset of target chips. Find the table that contains your target microprocessor. Within the table, in the 1st column find the library index file you plan to use. In the 2nd column you will find the library modules which must be replaced.

The following steps will update existing library index files. If you want to save the original index files, you should first copy the runtime library directory, then set your working directory to that directory. Otherwise, set your working directory to the runtime library. We will start with the 68000 libraries (lib000, lib000.nf, lib000.l, lib000.lnf).

1. Recompile all the C source files in the library, using -ss libdata -sd. This makes all library data separate in segment libdata. Execute a command like this for each c file:

   c68000 abort.c -ss libdata -sd

2. Pre-link each object module. Execute a command like this for each .ol file you just created:

   llink strcpy.ol -lo -w -o strcpy.ln

3. Update the library index file. Table 7-1 contains the library index files for target MC680000. Looking up lib000 in the table we find the following modules must be replaced: adln.ln. adlog.ln, adsqrt.ln, dpfncs.ln, dpopns.ln, fpopns.ln, pmain.ln, and xlfncs.ln. Execute the following commands:

   For the PC:

   libr -L lib000
   -d adln.ln adlog.ln adsqrt.ln dpfncs.ln
   dpopns.ln fpopns.ln pmain.ln xlfncs.ln

```
    -a adlnx.ln adlogx.ln adsqrtx.ln dpfncsx.ln
    dpopnsx.ln fpopnsx.ln pmainx.ln xlfncsx.ln

libr -L lib000 -u
```

For Unix hosts:

```
libr -L lib000 \
    -d adln.ln adlog.ln adsqrt.ln dpfncs.ln \
    dpopns.ln fpopns.ln pmain.ln xlfncs.ln
    -a adlnx.ln adlogx.ln adsqrtx.ln dpfncsx.ln \
    dpopnsx.ln fpopnsx.ln pmainx.ln xlfncsx.ln

libr -L lib000 -u
```

This replaces the assembly language modules with the "no-A5" versions. All the "no-A5" versions have names that end in "x". It also updates the library with the recompiled versions of the C modules.

Now we continue with the "no-floats" library.

4. Recompile xprintf and xscanf. Here you need the extra command line option -P NO_FP_IO to generate the "no-floats" version of xprintf and xscanf:

```
c68000 xprintf.c xscanf.c -P NO_FP_IO -ss libdata -sd
```

5. Pre-link xscanf and xprintf.Execute the following commands. Note the output going to the special .in suffix:

```
llink xprintf.ol -lo -w -o xprintf.in
llink xscanf.ol -lo -w -o xscanf.in
```

6. Update the no-floats library index file. This is similar to step 3), except that the only "no-A5" assembler module in the no-floats library is pmain:

```
libr -L lib000.nf -d pmain.ln -a pmainx.ln
libr -L lib000.nf -u
```

Now we continue with the long integer libraries. Here the procedure is similar, except in the following ways. An additional switch, -L is required on all compilations. The list of "no-A5" assembler modules is different. The library index file names are different. The linked module suffix names .ln and .in are replaced with .lln and .iln.

7. Recompile all the C source files in the library, using -sd, -ss libdata and -L . Execute a command like this for each c file:

```
c68000 abort.c -L -ss libdata -sd
```

8. Pre-link each object module same as step 2), except direct the output to .lln, not .ln, for example:

```
llink strcpy.ol -lo -w -o strcpy.lln
```

9. Rebuild the library index files. Again refer to Table 7-1 to find which modules must be replaced in library index file lib000.l. Execute these commands:

For the PC:

```
libr -L lib000.l -d adlnl.ln adlogl.ln
        adsqrtl.ln dpfncs.ln dpopns.ln
        fpopns.ln pmain.ln xlfncs.ln
```

```
        -a adlnlx.ln adloglx.ln
        adsqrtlx.ln dpfncsx.ln dpopnsx.ln
        fpopnsx.ln pmainx.ln xlfncsx.ln

    libr -L lib000.l -u
```

For Unix hosts:

```
    libr -L lib000.l -d adlnl.ln adlogl.ln \
        adsqrtl.ln dpfncs.ln dpopns.ln \
        fpopns.ln pmain.ln xlfncs.ln
        -a adlnlx.ln adloglx.ln \
        adsqrtlx.ln dpfncsx.ln dpopnsx.ln \
        fpopnsx.ln pmainx.ln xlfncsx.ln

    libr -L lib000.l -u
```

This replaces the assembly language modules with the "no-A5" versions. It also updates the library with the recompiled versions of the C modules. Now we continue with the long integer, no-floats library. Note that the list of assembly language modules is different here than it was in the short integer library. For example, lib000.l uses adsqrtl.ln whereas lib000 uses adsqrt.ln. Again, refer to Table 7-1 to find which modules must be replaced in library index file lib000.l.

10. Recompile no-floats, long integer version of xprintf and xscanf:

```
c68000 xprintf.c xscanf.c -P NO_FP_IO -ss libdata -L -sd
```

11. Pre-link xscanf and xprintf. This is the same as step 5), except direct the output to .iln instead of .in, that is:

```
llink xprintf.ol -lo -w -o xprintf.iln
llink xscanf.ol -lo -w -o xscanf.iln
```

12. Update the no-floats library index file:

```
libr -L lib000.lnf -d pmain.ln -a pmainx.ln
libr -L lib000.lnf -u
```

If you are using a C++ library (cpp000.lib or cpp020.lib), it will need to be updated as well. Remember that for MC68000, MC68EC000, MC68008, MC68010, and MC68302 targets the cpp000.lib library is used; otherwise the cpp020.lib library is used. Also note that since C++ assumes -L, a C long library must also be created as shown above. This example includes a rebuild of cpp000.lib.

1. Recompile all C++ source files using -sd and -ss libdata. Execute a command like this for each C++ file:

```
cp68000 array_del.cpp -sd -ss libdata -I. ..\cppinc
--exceptions --building_runtime
```

The cpp020.lib library was oringinally built using the MC68332 compiler, but other (non-68000) compilers could be used for this purpose.

2. Recompile the C file whatami.c:

```
c68000 whatami.c -sd -ss libdata
```

3. Pre-link each object module. Execute a command like this for each .ol file for just created:

```
llink array_del.ol -lo -w -o array_del.0ln
```

The .0ln extension is used for cpp000.lib and the .2ln extension is used for cpp020.lib.

4. Update the library index file:

   libr -L cpp000.lib -u

   The procedure is the same for every other target-specific library, except that the library index files are different and the list of assembler modules containing A5 references is different.

   The following table summarizes all the assembler modules that use A5-relative addressing and what library index files they belong to. In all cases the name of the replacement no-A5 version is formed by adding an "x" before the suffix. For example, the no-A5 replacement for adsqrt.ln is adsqrtx.ln.

   Here is how you use this table. Look up the library you are are rebuilding on the left-hand column. Update the library index file by deleting the named modules and adding their replacements. There is one table for each library directory which covers all the library index files in that directory.

Libraries		**Modules to be Replaced with no-A5 Versions**		
lib000	lib010	adln.ln	adlog.ln	adsqrt.ln
lib302	lib020s	dpfncs.ln	dpopns.ln	fpopns.ln
lib030s		pmain.ln	xlfncs.ln	
lib000.nf	lib000.lnf	pmain.ln		
lib010.nf	lib010.lnf			
lib302.nf	lib302.lnf			
lib020s.nf	lib030s.lnf			
lib000.l	lib010.l	adlnl.ln	adlogl.ln	adsqrtl.ln
lib302.l	lib020s.l	dpfncs.ln	dpopns.ln	fpopns.ln
lib030s.l		pmain.ln	xlfncs.ln	
lib302ap		adln.ln	adlog.ln	adsqrt.ln
lib302at		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn302a.ln	xlfncs.ln	
lib302ap.nf	lib302ap.lnf	pmn302a.ln		
lib302at.nf	lib302at.lnf			
lib302ap.l		adlnl.ln	adlogl.ln	adsqrtl.ln
lib302at.l		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn302a.ln	xlfncs.ln	
lib000r		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmainr.ln	xlfncs.ln	
lib000r.nf		pmainr.ln		
lib000r.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmainr.ln	xlfncs.ln	

Table 7-1: Library index files (MC68000, MC68010, MC68302)

Libraries	**Modules to be Replaced with no-A5 Versions**

lib020s	lib030s	adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmain.ln	xlfncs.ln	
lib020s.nf	lib020s.lnf	pmain.ln		
lib030s.nf	lib030s.lnf			
lib020s.l	lib030s.l	adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmain.ln	xlfncs.ln	
lib332		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn332.ln	xlfncs.ln	
lib332.nf	lib332.lnf	pmn332.ln		
lib332.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn332.ln	xlfncs.ln	
lib340		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn340.ln	xlfncs.ln	
lib340.nf	lib340.lnf	pmn340.ln		
lib340.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn340.ln	xlfncs.ln	
lib340b		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn340b.ln	xlfncs.ln	
lib340b.nf	lib340b.lnf	pmn340b.ln		
lib340b.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn340b.ln	xlfncs.ln	
lib360		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn360.ln	xlfncs.ln	
lib360.nf	lib360.lnf	pmn360.ln		
lib360.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn360.ln	xlfncs.ln	
lib360b		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn360b.ln	xlfncs.ln	
lib360b.nf	lib360b.lnf	pmn360b.ln		
lib360b.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn360b.ln	xlfncs.ln	
lib332r		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln

		pmn332r.ln	xlfncs.ln	
lib332r.nf	lib332r.lnf	pmn332r.ln		
lib332r.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn332r.ln	xlfncs.ln	
lib030r		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn030r.ln	xlfncs.ln	
lib030r.nf	lib030r.lnf	pmn030r.ln		
lib030r.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmn030r.ln	xlfncs.ln	

Table 7-2: Library index files (MC68020, MC68030, MC68332, MC68340, MC68360), no 68881/68882

Libraries		**Modules to be Replaced with no-A5 Versions**		
lib020h	lib030h	acos.ln	asin.ln	log.ln
		log2.ln	log10.ln	pmain.ln
		sqrt.ln		
lib020h.nf	lib020h.lnf	pmain.ln		
lib030h.nf	lib030h.lnf			
lib020h.l	lib030h.l	acosl.ln	asinl.ln	logl.ln
		log2l.ln	log10l.ln	pmain.ln
		sqrtl.ln		
lib030hr		acos.ln	asin.ln	log.ln
		log2.ln	log10.ln	pmn030r.ln
		sqrt.ln		
lib030hr.nf	lib020hr.lnf	pmn030r.ln		
lib030hr.l		acosl.ln	asinl.ln	logl.ln
		log2l.ln	log10l.ln	pmn030r.ln
		sqrtl.ln		

Table 7-3: Library index files (MC68020, MC68030), with 68881/68882

Libraries		**Modules to be Replaced with no-A5 Versions**		
libe40		adln.ln	adlog.ln	adsqrt.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmain.ln	xlfncs.ln	
libe40.nf	libe40.lnf	pmain.ln		
libe40.l		adlnl.ln	adlogl.ln	adsqrtl.ln
		dpfncs.ln	dpopns.ln	fpopns.ln
		pmain.ln	xlfncs.ln	

Table 7-4: Library index files (MC68EC040)

Libraries		Modules to be Replaced with no-A5 Versions		
lib040		acos.ln	asin.ln	log.ln
		log2.ln	log10.ln	pmain.ln
		sqrt.ln		
lib040.nf	lib040.lnf	pmain.ln		
lib040.l		acosl.ln	asinl.ln	logl.ln
		log2l.ln	log10l.ln	pmain.ln
		sqrtl.ln		

Table 7-5: Library index files (MC68040)

Libraries		Modules to be Replaced with no-A5 Versions		
lib060		acos.ln	asin.ln	log.ln
		log2.ln	log10.ln	pmainf.ln
		sqrt.ln		
lib060.nf	lib060.lnf	pmain.ln		
lib060.l		acosl.ln	asinl.ln	logl.ln
		log2l.ln	log10l.ln	pmainf.ln
		sqrtl.ln		

Table 7-6: Library index files (MC68060)